

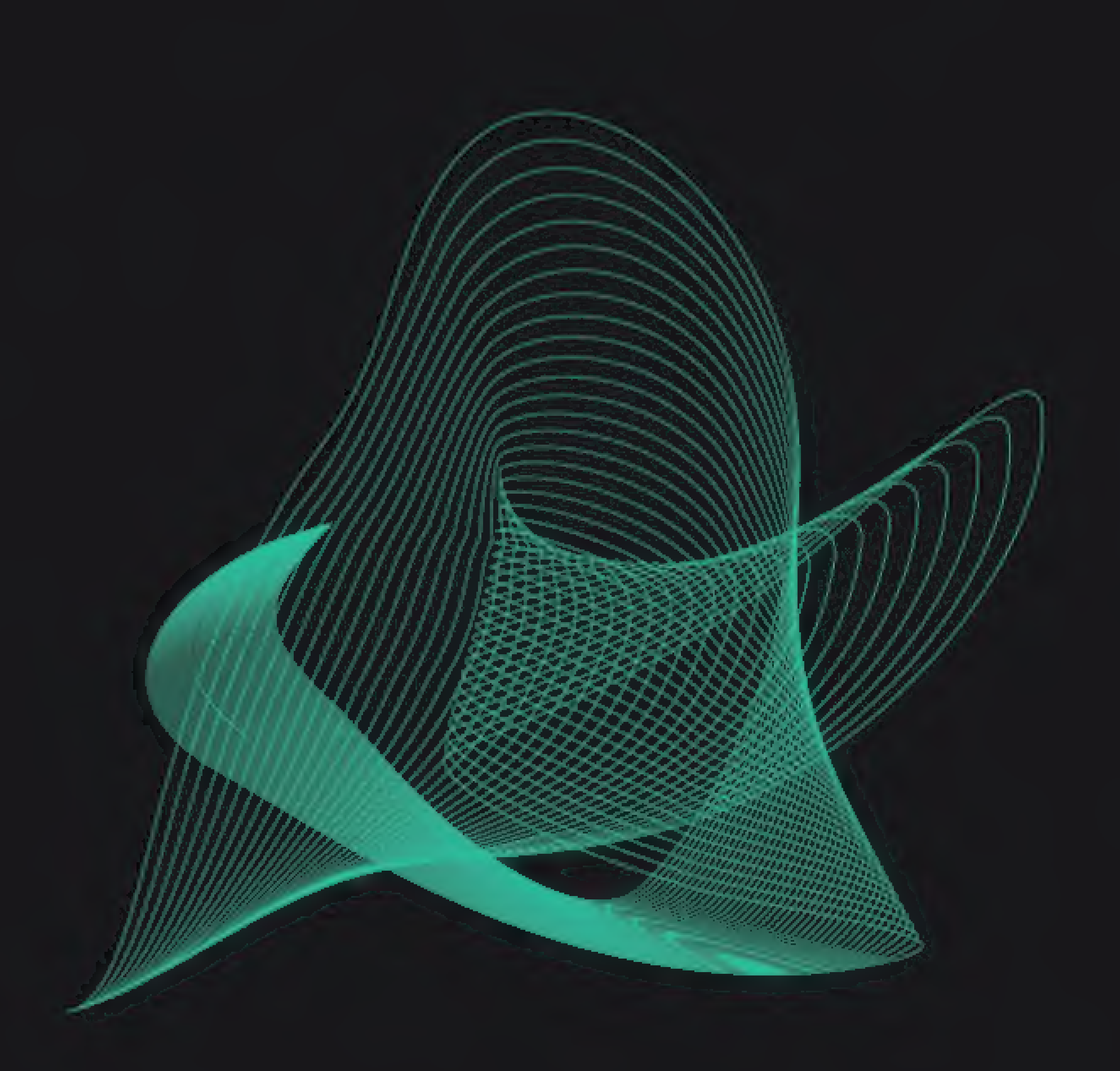
21世纪高等学校信息安全专业规划教材

网络安全程序设计

李红娇 ◎ 主　编

李晋国　李婧 ◎ 副主编

顾春华 ◎ 主　审



清華大學出版社

21世纪高等学校信息安全专业规划教材

网络安全程序设计

李红娇　主　编
李晋国　李　婧　副主编
顾春华　主　审

清华大学出版社
北京

内 容 简 介

本书以网络安全程序设计基础和主要技术为核心内容。全书共 8 章，主要内容包括：第 1 章是网络空间安全学科相关介绍；第 2 章是网络安全编程基础，包括 Socket 编程与 VC++网络安全编程基础；第 3～8 章是全书的重点，介绍密码学编程，基于 OpenSSL 开发包的网络安全编程，网络扫描器设计，防火墙设计与实现，入侵检测模型的设计与实现以及应用系统安全编程。本书内容丰富、实用，涵盖网络安全程序设计的基本核心技术，并给出了代码实例，有助于读者理论结合实际地理解掌握网络安全程序设计技术。

本书可作为普通高等院校信息安全专业、计算机科学与技术专业、电子信息专业本科生和研究生的教材，也可供相关行业从业人员学习参考。

图书在版编目(CIP)数据

网络安全程序设计/李红娇主编. —北京：清华大学出版社，2017
(21 世纪高等学校信息安全专业规划教材)
ISBN 978-7-302-45180-8

Ⅰ. ①网…　Ⅱ. ①李…　Ⅲ. ①计算机网络－网络安全－程序设计－高等学校－教材　Ⅳ. ①TP311.1

中国版本图书馆 CIP 数据核字(2016)第 239586 号

责任编辑：魏江江　薛　阳
封面设计：
责任校对：焦丽丽
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　张**：19.25　　**字　　数**：469 千字
版　　次：2017 年 3 月第 1 版　　**印　　次**：2017 年 3 月第 1 次印刷
印　　数：1～　000
定　　价：　.00 元

产品编号：068241-01

前　言

随着互联网应用的日益广泛，围绕网络信息的获取、使用、传输引发的安全问题越来越显得重要，网络空间安全也上升为国家战略，实践能力是网络空间安全创新人才培养的核心之一。本书是为高等学校的本科生、研究生提供的网络安全程序设计教材。

网络空间安全涉及数学、计算机科学与技术、信息与通信工程等多个学科，已形成了一个相对独立的教学和研究领域。网络安全程序设计对学生的要求相对比较高，需要高级语言编程、操作系统、计算机网络、密码学与信息安全等基础知识以及一些工具软件的应用。

本书从网络空间安全的必要性以及对创新人才培养的需求出发，阐述网络安全程序设计的编程基础与核心技术，对每个技术的讲述包括基本概念、基本原理及编程实例，将基础知识与编程实践结合，这对启发学生的思考以及提升动手能力是十分重要的。从而学生更能深入理解每种安全机制的实质，也有助于学生理论联系实际地根据实际应用掌握网络安全编程技术。

全书共 8 章。第 1 章概要介绍网络空间安全的必要性、网络空间安全对人才培养的新要求以及网络安全程序设计相关知识；第 2 章介绍网络安全编程基础，包括 Socket 编程以及 VC++网络安全编程；第 3 章阐述密码学基础知识，基于经典密码算法的安全编程实例；在此基础上，第 4 章讲述基于网络安全开发包 OpenSSL 的编程实践；第 5 章介绍网络扫描器的设计，包括 ICMP 扫描、TCP 扫描、木马扫描等基本原理与编程实现；第 6 章介绍了防火墙技术以及基于包过滤技术的防火墙实现；第 7 章介绍入侵检测系统原理、技术与实现；第 8 章介绍两种实际应用系统编程，包括基于 OpenSSL 的安全 Web 服务器设计实现及安全电子邮件编程。

本书的建议学时为 48 学时，其中课堂讲解部分为 24 学时，上机实验 24 学时。根据各专业的不同教学需求，以上学时安排和内容可根据实际需要进行调整。

本书由李红娇担任主编与统稿工作，李晋国、李婧担任副主编。李红娇负责编写第 1～4 章。第 5 章和第 6 章内容由李婧负责编写，第 7 章和第 8 章内容由李晋国负责编写。许智、陈晶晶、郭政伟参与了本书的编辑及程序代码调试工作。本书由上海电力

学院顾春华教授主审。本教材编写得到了上海市信息安全管理重点实验室开放课题(编号AGK2015005)以及上海市科委地方能力建设项目(No. 15110500700)资助。在编写过程中,也参考了一批技术文献、著作、教材及网络资源,为本书的编写奠定了宝贵的基础,在此一并表示衷心的感谢。

由于编者水平有限,书中难免有疏漏和不足之处,恳请专家和读者批评指正。

编　者

2016年12月于上海

目　　录

第1章　绪　　论

信息技术和应用的不断发展变化，给网络空间安全带来了巨大挑战，维护网络空间安全已经成为国家安全的战略高地，国家高度重视网络空间安全人才培养，增设网络空间安全一级学科，提高学生的实践能力是培养网络安全创新人才的核心之一。因此，必须掌握网络安全程序设计基础知识。

1.1　网络空间安全的必要性

1.1.1　技术层面

20 世纪 60 年代开始，美国国防部的高级研究计划局(Advance Research Projects Agency，ARPA)开始建立 ARPANet，即因特网的前身。因特网的迅猛发展始于 20 世纪 90 年代，由欧洲原子核研究组织 CERN 开发的万维网 WWW 被广泛使用在因特网上，大大方便了广大非网络专业人员对网络的使用，成为因特网用户指数级增长的主要驱动力。今天的因特网已不再是计算机人员和军事部门进行科研的领域，而是变成了一个开发和使用信息资源的覆盖全球的信息海洋，覆盖了社会生活的方方面面，构成了一个信息社会的缩影。目前，互联网正从 IPv4 向 IPv6 跨越。然而因特网也有其固有的缺点。

(1) 因特网是一个开放的、无控制机构的网络，黑客(Hacker)经常会侵入网络中的计算机系统，或窃取机密数据和盗用特权，或破坏重要数据，或使系统功能得不到充分发挥直至瘫痪。

(2) 因特网的大多数数据传输是基于 TCP/IP 通信协议进行的，这些协议缺乏使传输过程中的信息不被窃取的安全措施。

(3) 因特网上的通信业务多数使用 UNIX 操作系统来支持，UNIX 操作系统中明显存在的安全脆弱性问题会直接影响安全服务。

(4) 在计算机上存储、传输和处理的电子信息，还没有像传统的邮件通信那样进行信封保护和签字盖章。信息的来源和去向是否真实，内容是否被改动，以及是否泄漏等，在应用层支持的服务协议中是凭着君子协定来维系的。

(5) 电子邮件存在着被拆看、误投和伪造的可能性。使用电子邮件来传输重要机密信息会存在很大的危险。

(6) 计算机病毒通过因特网的传播给上网用户带来极大的危害，病毒可以使计算机和计算机网络系统瘫痪、数据和文件丢失。在网络上传播病毒可以通过公共匿名 FTP 文件传送，也可以通过邮件和邮件的附件传播。

安全性问题成为困扰因特网用户发展的一个主要因素。计算机病毒、网络蠕虫的广泛传播，计算机网络黑客的恶意攻击，DDOS 攻击的强大破坏力、网上窃密和犯罪的增多，使得网络安全问题关系到未来网络应用的深入发展。当信息技术快速步入网络时代，跨地域、跨

管理域的协作不可避免，多个系统之间存在频繁交互或大规模数据流动，专一、严格的信息控制策略变得不合时宜，信息安全领域随即进入了以立体防御、深度防御为核心思想的信息安全保障时代，形成了以预警、攻击防护、响应、恢复为主要特征的全生命周期安全管理，出现了大规模网络攻击与防护、互联网安全监管等各项新的研究内容。安全管理也由信息安全产品测评发展到大规模信息系统的整体风险评估与等级保护等。因此，开始针对信息安全体系进行研究，重在运行安全与数据安全，兼顾内容安全。

尽管当前信息安全技术得到了很大的发展，但是，信息技术和应用的不断发展变化也给其带来了巨大挑战，这些挑战主要有以下5个方面。

1. 新型通信网络

各国大力投入对新型通信网络的研究，欧盟FP7计划的Challenge One项目目标是提升网络灵活性以及可重构，日本AKARI计划主旨是网络虚拟化、多样化数据接入、网络功能扩展；美国国家科学基金会(National Science Foundation，NSF)的FIA项目以构建网络内容为导向、具备更安全表达性的网络为主旨；斯坦福大学的OpenFlow旨在构建网络控制平面与数据平面相分离的体系、实现灵活控制；软件定义网络(Software Define Network，SDN)由OpenFlow发展而来，被ITU等认可为新型通信网络的主流架构。随着新型通信网络技术的发展，国际电信联盟电信标准分局(International Telecommunication Union Telecommunication Sector，ITU-T)、国际互联网工程任务组(The Internet Engineering Task Force，IETF)、开放网络基金会(Open Networking Foundation，ONF)、欧洲电信标准化协会(European Telecommunication Standards Institute，ETSI)等正着手制定相应标准。

由于新型通信网络以用户为中心、异构、动态、虚拟、开放，网络业务需求具备应用异构性、系统可扩展性、需求动态性、服务客户化；新型通信网络的控制集中性导致安全威胁更集中、更开放，受安全威胁面更大，虚拟性导致攻击形式趋于复杂和动态；因此，新型通信网络拓扑的动态性，控制的开放性，流量的隔离性，资源的虚拟性对信息安全提出了新挑战：网络结构和安全行为关系难以准确描述，控制节点的脆弱性影响整个网络，难以对控制入侵行为进行分析，虚实资源的复杂映射导致威胁态势难以准确分析。一些重要的科学问题，如网络结构、脆弱分析、检测机理、安全态势等需要新的思路来解决。

2. 云计算

云计算的安全问题是用户不再对数据和环境拥有完全控制权，云计算的出现彻底打破了地域的概念，数据不再存放在某个确定的物理节点，而是由服务商动态提供存储空间，这些空间有可能是现实的，也可能是虚拟的，还可能分布在不同国家及区域，用户对存放在云中的数据不能像从前那样具有完全的管理权。

相比传统的数据存储和处理方式，云计算时代的数据存储和处理，对于用户而言，变得非常不可控，云环境中用户数据安全与隐私保护难以实现。传统模式下，用户可以对其数据通过物理和逻辑划分安全域实现有效的隔离和保护。在云计算环境下，各类云应用不再依靠机器或网络形成固定不变的基础设施物理边界和安全边界，数据安全由云计算提供商负责。云计算中多层服务模式同样存在安全隐患。云计算发展的趋势之一是IT服务专业化，即云服务商在对外提供服务的同时，自身也需要购买其他云服务商所提供的服务；用户所享用的云服务间接涉及多个服务提供商，多层转包无疑极大地提高了问题的复杂性，进一

步增加了安全风险；虚拟运算平台的安全漏洞不断涌现，直接威胁云安全根基；云端大量采用虚拟技术，虚拟平台的安全无疑关系到云体系的架构安全；虚拟运算平台变得越来越复杂和庞大，管理难度也随之增大，如果黑客利用安全漏洞获得虚拟平台的管理控制权，后果将不堪设想。

3. 大数据

随着互联网、移动互联网、社交网络、数码设备、物联网/传感器等技术的发展，各种设备产生的数据量将会急剧增长。根据互联网数据中心（Internet Data Center，IDC）预测，未来10 年内全球数据量将以超过 40％的速度增长，2020 年全球数据量将达到 35ZB。

大数据的概念在学术界由来已久，但真正进入公众视野是在 2011 年麦肯锡发布的研究报告——《大数据：创新、竞争和生产力的下一个新领域》以后。普遍的观点认为，大数据是指规模大且复杂，以至于很难用现有数据库管理工具或数据处理方法来处理的数据集。大数据的常见特点包括大规模（volume）、高速性（velocity）和多样性（variety）。根据来源的不同，大数据大致可分为如下几类。

（1）来自于人。人们在互联网活动以及使用移动互联网过程中所产生的各类数据，包括文字、图片、视频等信息。

（2）来自于机。各类计算机信息系统产生的数据，以文件、数据库、多媒体等形式存在，也包括审计、日志等自动生成的信息。

（3）来自于物。各类数字设备所采集的数据，如摄像头产生的数字信号、医疗物联网中产生的人的各项特征值、天文望远镜所产生的大量数据等。

大数据从概念走向实践，引发个人隐私安全问题。2011 年 4 月初，全球最大的电子邮件营销公司艾司隆（Epsilon）发生史上最严重的黑客入侵事件，导致许多企业客户名单以及电子邮件地址因此外泄。2011 年年底有网友爆料有黑客在网上公开了知名程序员网站CSDN 的用户数据库，2014 年年初携程网被怀疑储存用户信用卡信息存在泄漏风险。根据智能手机存储、显示的位置信息等多种数据组合，已可相对精准地锁定个人，用户个人隐私信息安全问题堪忧。

大数据时代国家安全将受到信息战与网络恐怖主义的威胁，大数据成为网络攻击的显著目标，并成为高级可持续攻击（Advanced Persistent Threat，APT）的载体，各国信息基础设施和重要机构都可能成为打击目标，而保护其免受攻击早已超出军事职权和能力范围，庞大海量的大数据涉及的方面之广，使得大数据也将为网络恐怖主义提供新的资源支持。

因其体量巨大、产生高速、类型多样、分布协同等特征，大数据面临严峻的信息安全挑战。传统的信息安全技术难以直接应用，发展一套全新的大数据系统安全理论和技术目前还不现实。因此，应采用现有安全技术，结合具体应用，利用新的思路，将大数据变成小数据，研究相关的安全关键技术，包括大数据中的用户隐私保护，大数据的可信性，大数据的访问控制技术，大数据可信度量技术，高效的大数据密码学以及针对不同结构的结构化、半结构化和非结构化数据，研究如何有效地进行安全管理、访问控制和安全通信。此外，在多租户的模式下，需要在保证效率的前提下，实现租户数据的隔离性、保密性、完整性、可用性、可控性和可追踪性。

4. 物联网与可穿戴设备

物联网的广泛应用将规避因特网应用上的局限性与安全性问题，通过射频识别（Radio

Frequency Identification，RFID）、红外感应器、全球定位系统、激光扫描器等信息传感设备，按约定的协议，把特定区域里的任何物品与虚拟网络连接起来，进行信息交换和通信，以实现智能化识别、定位、跟踪、监控和管理。物联网实质上是传感网与因特网、移动通信网，“三网”高效融合的产物，建立本地化的相对保密的传感网络与物联网络，可提升本土信息流通的安全性。国家的各个关键部门、产业领域以及一些关键性基础设施的控制系统逐步实现网络化，可增强在国际信息竞争中的话语权，为解决信息安全问题提供方案。

物联网感知层的电子标签和传感网络节点资源有限——存储空间、计算资源、通信能力、运算速度有限，难以采用复杂的安全机制，给传统的密码学和信息安全提出了挑战；感知层采用无线通信——传递信息暴露于大庭广众之下，给攻击者带来更多机会；物联网系统对应用完全开放将带来更多安全隐患。

可穿戴设备的隐蔽性和智能尘埃（intelligent mote）电子标签不可见给用户隐私保护带来极大的困难；谷歌眼镜和普通眼镜直观上难于区别，但却能拍摄尺寸极小的电子标签（尺寸为长 0.1mm，宽 0.1mm，厚 0.01mm），用户很难在物理上发现已经被跟踪，因此，保护用户隐私难度更大；认证过程需要物品的身份和位置信息，这加大了隐私保护难度。

物联网应用层信息安全问题来自物联网的安全体系架构带来的挑战。物联网中数亿计的设备接入，海量的数据信息，大量异构网络的存在，大规模的分布式应用系统，使物联网的安全体系架构面临着更加艰巨的挑战；物联网的访问控制存在难点，由于物联网部署的可扩展性、移动性和复杂性，使得对物品的访问控制很难有效地进行；物品间集群概念的引入，还需要解决群组认证的问题；物联网网络态势感知与评估理论和技术需求迫切，如何从大数据中升华智慧，对大规模物联网正常运转进行全面的态势感知和安全评估，以保障其安全运行和故障报警是正在开展的研究热点。

5. 量子网攻

美国《纽约时报》曝光的“量子”项目让舆论大吃一惊——美国国家安全局（National Security Agency，NSA）能够将一种秘密技术成功植入没有联网的计算机，对其数据进行任意更改。NSA 至少从 2008 年就开始使用这项名为“高科技广播频率”（High tech radio frequency technology）的技术，并利用该技术成功入侵了全球近十万台计算机。一般来说，计算机间谍软件都是通过网络进行传播、植入的，但据悉 NSA 已经开始使用一种可以在计算机不接入互联网的情况下接入并修改其中数据的秘密技术。NSA 所使用的其中一件装备就是外形同普通 USB 设备无异的 Cottonmouth，只是该装置内嵌一个微型发射-接收器。

值得注意的是，在 2008—2010 年夏天美国对伊朗核设施采取的网络攻击中，美国就利用了这项技术向伊朗核设施植入“震网”病毒，这也是该技术第一次参与实战，据《纽约时报》透露，美国还出于反恐目的在沙特阿拉伯、印度和巴基斯坦网络中植入了这一间谍软件。

1.1.2 网络安全与国家战略

21 世纪是信息的时代，信息成为重要的战略资源。信息技术改变着人们的生活和工作方式。社会对计算机和网络的依赖越来越大。敌对势力的破坏、恶意软件的攻击等已对计算机和网络系统的安全构成极大的威胁，如果计算机和网络系统的安全受到破坏，不仅会造

成巨大的经济损失，甚至会导致社会混乱。信息安全关系到国家安全、社会稳定、经济发展、人民生活等各个方面，必须确保我国的信息安全。要建设国家信息安全保障体系，政府、军队和企业都需要大量信息安全专门人才。

1. 电子政务、电子党务对信息安全的需求

电子政务网、电子党务网是政府与党务办公、联系社会、服务社会的关键基础设施。政务网、党务网上的信息涉密程度高，对信息的保密性、完整性和可用性的要求也较高。电子政务、电子党务信息网络的建设和维护需要一支具有较高信息安全专业水平的建设和管理队伍。

2. 国防建设对信息安全的需求

信息对抗的攻防能力已成为国防力量之一，对网络的攻击也是一种威慑力量。美国很早就提出了信息战的概念，在"海湾战争"期间美军成功地对伊拉克发动了信息战。2009年6月美国国防部长签署命令，正式成立美军的网络司令部。我国也应有自己的网络安全队伍，保障关系国计民生的重要网络信息系统的安全，防范可能的入侵和攻击，并具有必要的信息对抗能力，这些都需要大量的高级信息安全专业人才。

3. 维护公共安全对信息安全的需求

近年来，各种形式的网络犯罪给全球不少国家都带来了巨大损失。美国政府公布的一份国家安全报告认为，21世纪对美国国家安全威胁最严重的是网络恐怖主义。美国中央情报局成立了一个专门负责研究遏制计算机犯罪的信息技术中心。为了遏制各种形式的网络犯罪，公安部门应当有能力通过合法监听得到通信内容；对于所得到的特定内容应当能知道其来源与去向；在必要的条件下能控制特定信息的传播。除了网络恐怖之外，一些网站传播低俗的内容，严重危害青少年的身心健康。要阻止网络犯罪和传播低俗内容，需要组建专门的网络警察队伍。因此，需要大量高素质的信息安全专业人才。

4. 企业发展对信息安全的需求

企业在利用信息化优势发展的过程中，需要把自身的网络安全风险降到最低。这就必须要求有足够的信息安全人才，企业对信息安全人员的需求不但在数量上迅速增长，而且对相关职位人才的任职能力都提出了更高的要求。国家已经颁布了《信息安全等级保护管理办法》，对企业配备信息安全人员做出了硬性规定。

5. 个人用户对信息安全的需求

个人用户在信息安全方面有通信内容机密性、用户信息隐私性、应用系统可信性等需求。这就要求信息服务提供商要能满足用户的安全需求，也需要大量专门的信息安全人才。

我国高度重视网络安全工作。2014年2月27日，中央成立网络安全与信息化领导小组，习近平总书记亲自担任组长。习总书记在第一次会议上强调指出："网络安全和信息化是事关国家安全和国家发展、事关广大人民群众生活的重大战略问题，要从国际国内大势出发，总体布局，统筹各方，创新发展。"网络安全和信息化是一体之两翼、双轮之驱动，必须统一谋划、统一部署、统一推进、统一实施。"没有网络安全，就没有国家安全；没有信息化，就没有现代化。"这一科学论断阐述了网络安全与国家信息化之间的紧密关系，使我们认识到

网络安全为国家信息化建设提供安全保障的极端重要性。习总书记的重要讲话精神，为我们做好网络空间安全学科专业建设注入活力，极大地增强了我们做好网络空间安全学科的信心。网络安全已成为国家安全的重要组成部分，要从根本上提高我国网络安全水平，健全网络空间安全保障体系，必须培养高素质的网络空间安全专业人才。

没有网络安全就没有国家安全。网络安全是一个关系到国家安全和社会稳定的重要问题。其重要性正随着全球信息化的步伐与日俱增。在我国，国家高度重视网络空间安全保障工作，网络信息安全上升至国家战略。2013 年 11 月 12 日，中国共产党中央国家安全委员会成立。2014 年 2 月 2 日，中央网络安全和信息化领导小组成立，习近平指出网络安全和信息化是事关国家安全和国家发展、事关广大人民群众工作生活的重大战略问题。2015 年 1 月 23 日，中共中央政治局召开会议，审议通过《国家安全战略纲要》，指出要做好各领域国家安全工作，大力推进国家安全各种保障能力建设，把法治贯穿于维护国家安全的全过程。2015 年 4 月 20 日，《国家安全法(草案)》二审稿增加了国家"建设国家网络与信息安全保障体系，提升网络与信息安全保护能力""维护国家网络空间主权"的规定。2015 年 7 月 1 日，第十二届全国人民代表大会常务委员会第十五次会议通过新的国家安全法。

1.2 网络空间安全学科研究的主要内容

网络空间(Cyberspace)是通过全球互联网和计算系统进行通信、控制和信息共享的动态(不断变化)虚拟空间，在信息时代是社会有机运行的神经指挥系统，目前已经成为继陆、海、空、太空之后的第 5 空间。在网络空间里不仅包括通过网络互联的各种计算系统(包括各种智能终端)、连接端系统的网络、连接网络的互联网和受控系统，也包括其中的硬件、软件乃至产生、处理、传输、存储的各种数据或信息。

与其他空间不同的特点是，网络空间没有明确的、固定的边界，也没有集中的控制权威。而网络空间安全(Cyberspace Security，CS)研究网络空间中的安全威胁和防护问题，即在有敌手(Adversary)的对抗环境下，研究信息在产生、传输、存储、处理的各个环节中所面临的威胁和防御措施，以及网络和系统本身的威胁和防护机制。网络空间安全不仅包括传统信息安全所研究的信息的保密性、完整性和可用性，同时还包括构成网络空间基础设施的安全和可信性。这里，需要明确信息安全、网络安全、网络空间安全概念的异同，三者均属于非传统安全，均聚焦于信息安全问题。网络安全、网络空间安全的核心是信息安全问题，只是出发点和侧重点有所差别。信息安全使用范围比较广，可以指线下和线上的信息安全，既可以指称传统的信息系统安全，也可以指网络安全和网络空间安全，但无法完全替代网络安全与网络空间安全的内涵；网络安全可以指信息安全或网络空间安全，但侧重点是线上安全和网络社会安全；网络空间安全可以指称信息安全或网络空间安全，但侧重点是与陆、海、空、太空并列的空间概念。网络安全、网络空间安全、信息安全三者相比较，前两者反映的信息安全更立体、更宽域、更多层次，也更多样，更体现网络和空间的特征，并与其他安全领域有更多的渗透与融合。

网络空间安全涉及数学、计算机科学与安全、信息与通信工程等多个学科，已形成了一个相对独立的教学和研究领域。建立网络空间安全一级学科的目标是，通过网络空间安全

学科的培养，帮助学生掌握密码和网络空间安全基础理论和技术方法，掌握信息系统安全、网络基础设施安全、信息内容安全和信息对抗等相关专门知识，并具有较高网络空间安全综合专业素质、较强的实践能力和创新能力，能够承担科研院所、企事业单位和行政管理部门网络空间安全方面的科学研究、技术开发及管理工作。

如图 1-1 所示，网络空间安全学科主要研究方向及内容如下。



图 1-1　网络空间安全学科体系

网络空间安全基础理论：为其他方向提供理论、架构和方法学指导；包括从事网络空间安全研究工作所需要的数学、物理、电工学、计算机科学与技术、信息论、控制论、博弈论、密码学理论、网络空间安全法律法规等基础理论。

物理安全，包括芯片安全防护、电子对抗、电子辐射的安全、物理层通信安全等方面的理论与技术。

系统安全，保证网络空间中单元计算系统安全、可信；包括软件安全、操作系统安全、分布式系统安全、虚拟化安全、可信计算、移动终端安全、工业控制系统安全等方面的理论与技术。

网络安全，保证连接计算机的网络自身安全和传输信息安全；包括各类无线通信网络、计算机网络、物联网等的安全协议，攻防对抗、网络安全管理、取证与追踪等方面的理论与技术。

数据与信息安全，包括隐私保护、数据存储与恢复、数字版权保护、舆情分析、垃圾信息识别与过滤等方面的理论与技术。

网络空间安全一级学科的理论方法和方法论基础，涉及数学、信息论、计算复杂理论、控

制论、系统论、认知科学、博弈论、管理学等。其学科方法论基础：信息安全学科有其独特的方法论，与数学或计算机科学等学科的方法论既有联系又有区别。包括观察、实验、猜想、归纳、类比和演绎推理，以及理论分析、设计实现、测试分析等，综合形成了逆向验证的方法论。沈院士表示，信息安全保障体系是一个复杂的系统，必须从复杂系统的观点，采用从定性到定量的综合集成的思想方法，追求整体效能。从系统工程方法论的观点出发，网络空间安全不能简单地采用还原论的观点处理，必须遵循“木桶原理”，注重整体安全。

1.3 网络空间安全对人才培养的新要求

1.3.1 我国网络空间安全面临的形势

目前，网络空间和网络空间安全成为社会公众关注的话题，网络空间安全人才培养体系更成为人们关注的焦点。

近年来，随着社会信息化的不断加深，各种信息安全风险也伴随而生。国际上围绕网络空间安全的斗争愈演愈烈，争夺网络空间安全控制权是战略制高点。我国的网络空间面临着来自外部的威胁，近年来发生的许多安全事件表明，我国已经在网络空间安全方面处于极为被动的局面。“斯诺登事件”给我们的经验和教训是广泛而深刻的，在网络空间，围绕信息安全的斗争是激烈的。“斯诺登事件”也给我国敲响了一个警钟，再也不能用一般的力量、行动和认识对待网络空间安全问题。“斯诺登事件”证明美国在网络空间方面是有规划、有计划、成体系地部署和构建攻击力量，动员全社会资源发展系列化的高技术手段以达到网络空间行动绝对自由的战略目的。“斯诺登事件”给我们的启示是，不能用一般力量来对付体系力量，应采取信息化条件下的体系对抗策略，积极构建网络空间安全学科体系，使我国网络信息安全人才成体系化、规模化、系统化培养，更好地满足国家安全对网络信息安全人才的需要。

我国已成为网络大国，由于网络技术基础薄弱和网络空间安全人才不足，我国还不是网络强国。网络安全关系到国家安全、社会稳定、经济发展、人民生活等各个方面，必须确保我国的信息安全，要建设国家信息安全保障体系，政府、军队、公安等国家重要部门，以及金融、电力、能源等重要基础设施等都需要大量信息安全专门人才。据不完全统计，截至 2014 年年底，我国重要行业信息系统和信息基础设施需要各类网络空间安全人才 70 万，预计到 2020 年，需要各类网络空间安全人才约 140 万人，而目前我国高等学校每年培养的信息安全相关人才不足 1.5 万人，远远不能满足网络空间安全的需要。

1.3.2 网络空间安全一级学科

网络空间安全人才培养是国家信息安全保障体系建设的基础和先决条件，网络安全学科建设则是高层次创新型信息安全人才培养的关键。网络空间安全人才培养是一个完整的社会系统工程，只有在一级学科目录规范下，才能按学士、硕士、博士成体系、全方位地培养国家需要的网络空间安全各类人才。

事实上，网络空间安全学科在我国经过十多年的发展，理论和技术已经较为成熟。主要体现在以下几个方面：一是网络空间安全学科具有明确的研究对象，并形成了相对独立的

理论体系和研究方法。研究对象是网络空间及其安全问题。网络空间安全问题随着互联网在社会各个领域的普及而更加突出。网络空间安全问题的突出特征是在复杂的国际社会环境下人与人的对抗。网络空间安全学科的理论基础主要包括数学、信息论和计算复杂性理论等，并已经形成了独立的基础理论体系、技术体系和应用体系。研究方法论包括基于数学困难问题的逻辑证明、基于博弈论的仿真计算和基于真实物理环境的实证分析三个核心内容。二是网络空间安全已经形成了若干相互关联的二级学科研究方向，密码学及应用、系统安全、网络安全是本学科多年来公认的三个比较成熟的研究领域，另外还包括网络空间安全理论基础和应用系统安全（比如社交网络、电子商务、内容安全、工控系统安全等）。三是网络空间安全的研究已得到国内外学术界的普遍认同，并已经有了多年的研究或信息安全相关专业人才培养的经验和基础。在我国，截至 2014 年，教育部批准全国共 116 所高校设置信息安全类相关本科专业，其中信息安全专业 87 个，信息对抗专业 17 个，保密管理专业 12 个，已经培养信息安全类专业本科毕业生约一万人/年。

信息安全学科建设历史可以追溯到 20 世纪 70 年代以前，那时国内只有少数专业性院校（如军事院校）设置了密码学学科。经教育部批准，1999 年西安电子科技大学等 4 所高校率先设置了信息对抗技术本科专业。2001 年，武汉大学首先正式设置信息安全本科专业。2002 年，在国务院学位委员会、教育部下发《关于做好博士学位授予一级学科范围内自主设置学科、专业工作的几点意见》后，信息安全学科发展迅速，北京理工大学、武汉大学等 43 所院校，分别挂靠在信息与通信工程、计算科学与技术、数学等一级学科，自主设立了信息安全相关二级学科。其中，设置信息安全二级学科的 18 个，设置网络信息安全二级学科的 6 个，设置信息对抗二级学科的 5 个，其他 14 个。2007 年年初，"教育部高等学校信息安全类专业教学指导委员会"成立，同年年底，教育部批准了 15 个学校的信息安全类专业为"国家特色专业建设点"。2003 年，武汉大学、华中科技大学、中国科学院软件所、国防科技大学等单位建立信息安全博士点。到目前为止设立信息安全本科专业的高校有近百所。

当初，设立信息安全一级学科是为了解决学科建设与人才培养不能满足信息安全保障体系建设的要求而提出的，目前尽管已有不少高校设置信息安全类本科专业，但在研究生培养专业目录中并没有与之相对应的信息安全学科。基础不同，方向各异，内容混乱，相互掣肘，严重影响信息安全人才有序培养，导致人才总量和结构远远不能满足需求，复合型人才和专业人才严重缺乏，严重影响我国信息安全自主创新能力，制约我国信息安全保障体系的建设，难以应对国际敌对势力的挑战。

2015 年 6 月 11 日，国内信息安全领域传来了一个振奋人心的消息：国务院学位委员会和教育部联合发出《关于增设网络空间安全一级学科的通知（学位[2015] 11 号）》，其中这样写道：为实施国家安全战略，加快网络空间安全高层次人才培养，根据《学位授予和人才培养学科目录设置与管理办法》的规定和程序，经专家论证，国务院学位委员会学科评议组评议，报国务院学位委员会批准，国务院学位委员会、教育部决定在"工学"门类下增设"网络空间安全"一级学科，学科代码为"0839"，授予"工学"学位。

2015 年 10 月 30 日，开启网络空间安全一级学科博士学位授权点的申报工作。

2016 年 1 月 28 日，国务院学位委员会发布《关于同意增列网络空间安全一级学科博士学位授权点的通知（学位[2016]2 号）》、《关于同意对应调整网络空间安全一级学科博士学位授权点的通知（学位[2016]2 号）》，同意 29 所高校增列网络空间安全一级学科博士学位

授权点。

建立网络空间一级学科是落实习近平总书记重要指示的重大举措，也是实现网络强国建设、应对网络空间复杂形势的迫切需要，打赢网络空间斗争之仗关键是人才。

1.3.3　网络空间安全创新人才培养体系

国家对网络空间安全人才的需求是全方位的，既包括网络空间安全战略人才，网络安全与信息科技领军人才，网络安全和信息技术工程人才，网络安全分析人才，也包括网络安全管理、运行维护人员和技能型操作使用人员等。网络空间安全创新人才培养并非只能研究型大学所为，一般的大专院校，甚至中专技校都有可能培养出创新性人才，即作为创新人才培养的关键的教育体系，应该赋予各种层次的学校以创新人才培养的任务。建立网络空间一级学科的目标是，通过网络空间安全学科的培养，帮助学生掌握密码和网络空间安全基础理论和技术方法，掌握信息系统安全、网络基础设施安全、信息内容安全和信息对抗等相关专门知识，并具有较高网络空间安全综合专业素质、较强的实践能力和创新能力，能够承担科研院所、企事业单位和行政管理部门网络空间安全方面的科学研究、技术开发及管理工作。

创新驱动，改革人才培养模式，人才需求极为迫切，必须以创新思路改革办学机制和模式。尤其是网络空间安全人才需要具备较强的实践能力。因此，当前，为做好网络空间安全创新人才培养体系建设，建议重点在实践方面做好以下工作。

(1) 把目前存在于不同学科下的信息安全相关专业划归到网络空间安全类(即网络空间安全类)，以便统筹管理。

制定网络空间安全类专业人才培养标准。制定网络空间安全一级学科博士和硕士学位基本要求、网络空间安全类本科专业教学质量国家标准、职业院校网络空间安全类专业教学标准。加强组织领导，统筹指导全国网络空间安全学科专业建设，在统筹、整合和扩充现有全国信息安全专业教学指导委员会职能基础上，成立国家级网络空间安全学科专业建设指导委员会，改变专业和学科切割局面，负责学科评议、本科和研究生一体化培养的指导，并组织教师培训、实践体系建设、教材建设，制定人才培养规范和指导意见。增设网络空间安全专业与学科方向对应的专业目录，扩大招生面，为满足社会大量网络信息安全人才需求和学科培养生源做支撑。

(2) 探索网络空间安全创新人才培养机制，学校与业界深度融合协同培养人才。

在网信办和教育部指导和支持下，有关高校要主动加强与行业主管部门、信息安全相关企事业单位、科研院所开展合作，共同制定网络空间安全人才培养目标，共同开设相关课程和编写教材，共同实施培养过程，共同评价培养质量。推动有关院校网络空间安全类专业积极参与教育部"卓越工程师教育培养计划"，促进网络空间安全人才培养与国家网络安全事业发展紧密结合，培养更多卓越网络空间安全人才。

(3) 强化网络空间安全实战技能培养和实习实训。

由网信办牵头组织有关高校规划建设一批网络空间安全实习实训基地，统筹安排网络空间安全类专业学生到国内信息安全机构参加实习实训。倡导国内信息安全企业与高校共同制定学生实习实训方案，主动接收学生开展实习实训。

(4) 建设开放实训平台，提高网络攻防实践能力，搭建基于网络的仿真模拟训练平台，

支持实验课程设计，构成网络攻防演练环境。

设立网络空间安全教育部重点实验室、国家重点实验室、国家工程实验室等科研平台，并在资金上给予大力扶持。组织好全国网络空间安全技能竞赛，激发学生创新积极性，提高实践攻关能力。

(5) 加强网络空间学科师资队伍建设。

网络空间安全学科是一个新兴交叉学科，做好人才培养工作，师资队伍是关键。在网信办和教育部指导下组织有关高校和军队科研院所有计划地对青年骨干教师进行业务培训；高校要聘请经验丰富的信息安全企事业单位、科研院所的网络安全科研和管理专家担任兼职教师。鼓励信息安全科研院所与高校科教合作、协同育人。加快教师队伍建设，提高教学质量，针对目前教师队伍弱的现状，制定培训计划，以"培训班""速成班""进修班"等不同模式进行全方位的教师培训，使大量非本学科老师转换学科方向，承担网络空间安全的教学任务。

1.4 网络安全程序设计基础知识

1.4.1 网络协议

1. OSI 参考模型

国际标准化组织(International Standard Organization，ISO)制定了开放系统互连(Open System Interconnection，OSI)参考模型作为理解和实现网络安全的基础。OSI 模型用途相当广泛，比如交换机、集线器、路由器等很多网络设备的设计都是参照 OSI 模型设计的。本节首先介绍 OSI 模型，在此基础上介绍 OSI 的安全体系结构。

1) 开放系统互连参考模型

开放式系统互连模型，一般称为 OSI 参考模型，是 ISO 组织在 1985 年研究的网络互连模型。国际标准化组织 ISO 发布的最著名的标准是 ISO IEC 7498，又称为 X. 200 协议。该体系结构标准定义了网络互连的 7 层框架。在这一框架下进一步详细规定了每一层的功能，以实现开放系统环境中的互连性、互操作性和应用的可移植性。开放系统 OSI 标准定制过程中所采用的方法是将整个庞大而复杂的问题划分为若干个容易处理的小问题，这就是分层体系结构方法。在 OSI 中，采用了三级抽象，即体系结构、服务定义和协议规定说明。

2) OSI 7 个层次划分原则

ISO 为了使网络应用更为广泛，推出了 OSI 参考模型。其目的就是推荐所有公司使用这个规范来控制网络。这样所有公司都有相同的规范，就能互连了。提供各种网络服务功能的计算机网络系统是非常复杂的。根据分而治之的原则，ISO 将整个通信功能划分为 7 个层次，划分原则如下。

(1) 网络中各节点都有相同的层次；

(2) 不同节点相同层次具有相同的功能；

(3) 同一节点内相邻层之间通过接口通信；

(4) 每一层使用下层提供的服务，并向其上层提供服务；

(5) 不同节点的同等层通过协议实现对等层之间的通信。

3) OSI 的 7 层协议模型

OSI 的 7 层协议模型如图 1-2 所示，每层的内容如下。

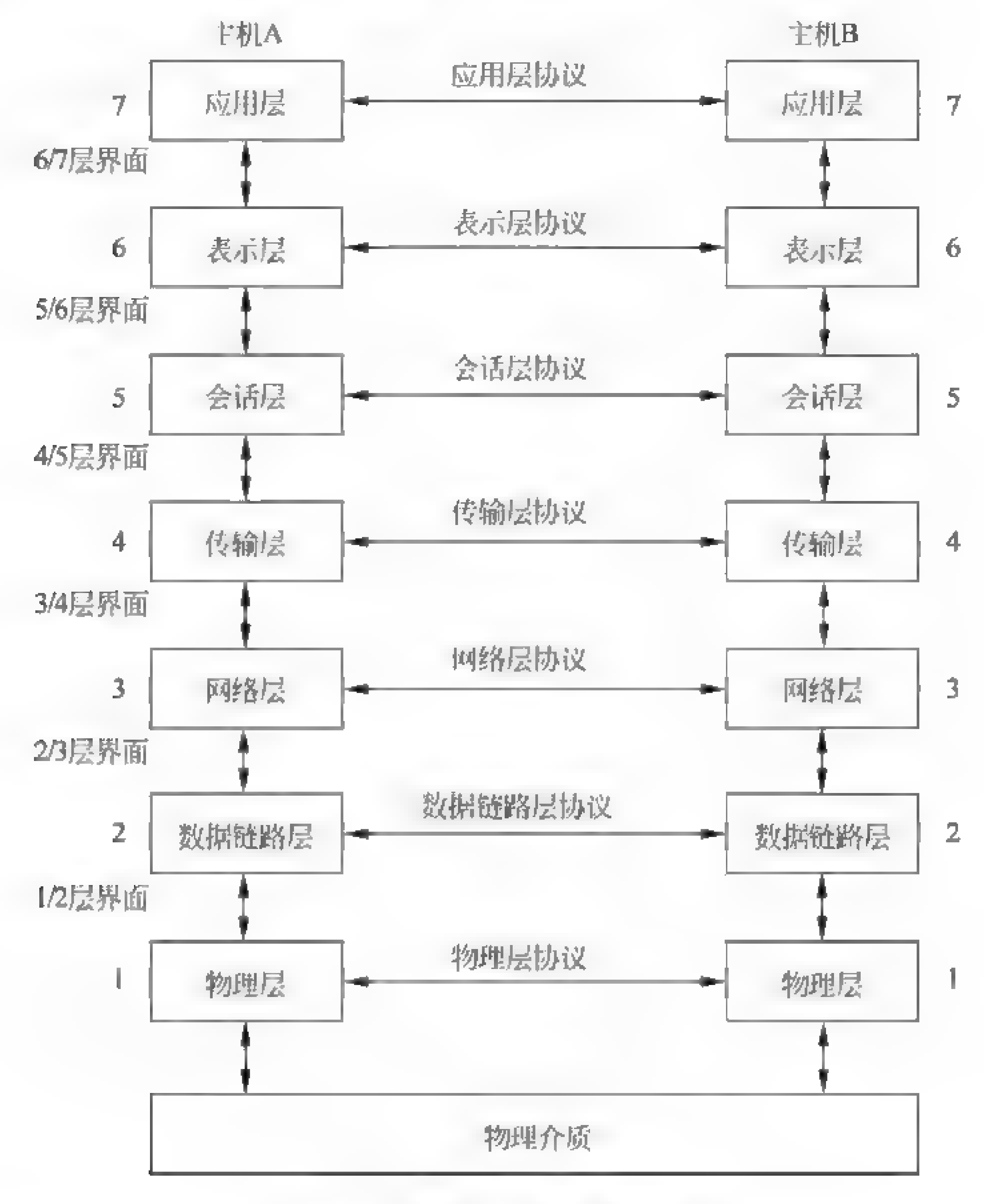


图 1-2　OSI 的 7 层协议模型

(1) 第 7 层应用层。应用层是 OSI 中的最高层。为特定类型的网络应用提供了访问 OSI 环境的手段。应用层确定进程之间通信的性质，以满足用户的需要。应用层不仅要提供应用进程所需要的信息交换和远程操作，而且还要作为应用进程的用户代理，来完成一些为进行信息交换所必需的功能。它包括：文件传送访问和管理、虚拟终端、事务处理、远程数据库访问、制造业报文规范、目录服务等协议。

(2) 第 6 层表示层。表示层主要用于处理两个通信系统中交换信息的表示方式。为上层用户解决用户信息的语法问题。它包括数据格式交换、数据加密与解密、数据压缩与恢复等功能。

(3) 第 5 层会话层。会话层在两个节点之间建立端连接。为端系统的应用程序之间提供了对话控制机制，决定通信是否被中断，以及通信中断时决定从何处重新发送。

(4) 第 4 层传输层。传输层完成常规数据递送 — 面向连接或无连接。为会话层用户提供一个端到端的可靠、透明和优化的数据传输服务机制。包括全双工或半双工、流控制和

错误恢复服务。

(5) 第3层网络层。网络层通过寻址来建立两个节点之间的连接,为源端的运输层送来的分组选择合适的路由和交换节点,正确无误地按照地址传送给目的端的运输层。它包括通过互联网络来路由和中继数据。

(6) 第2层数据链路层。数据链路层将数据分帧,并处理流控制。屏蔽物理层,为网络层提供一个数据链路的连接,在一条有可能出差错的物理连接上,进行几乎无差错的数据传输。本层指定拓扑结构并提供硬件寻址。

(7) 第1层物理层。物理层处于OSI参考模型的最底层。物理层的主要功能是利用物理传输介质为数据链路层提供物理连接,以便透明地传送比特流。

开放系统互连参考模型的基本构造技术是分层。每层的目的都是为上层提供某种服务,把这些层与提供服务的细节分开就形成结构化模型。在互连的开放系统中,各子系统的同一层共同构成开放系统中的一层,一般表示为 N 层——某一特定层; $N+1$ 层——相邻的高层; $N-1$ 层——相邻的低层。

在OSI参考模型中,对等实体的通信必须通过相邻低层以及下面各层通信来完成。从 $N+1$ 实体看,对等 $N+1$ 实体间的通信只能通过相邻对等 N 实体完成。N 实体向 $N+1$ 实体提供相互通信的能力称为 N 服务,即 $N+1$ 实体通过请求 N 服务完成对等实体通信。应注意的是,N 服务同时也要使用较低层提供的服务功能。

4) OSI 7层模型数据传输

数据发送时,从第7层传到第1层,接收数据则相反。上三层总称应用层,用来控制软件方面。下4层总称数据流层,用来管理硬件。数据在发至数据流层的时候将被拆分。每层封装后的数据单元叫法不同。上三层传输的数据统称为数据。在传输层的数据叫段,网络层叫包,数据链路层叫帧,物理层叫比特流,如图1-3所示为OSI 7层模型的数据传输示意图。

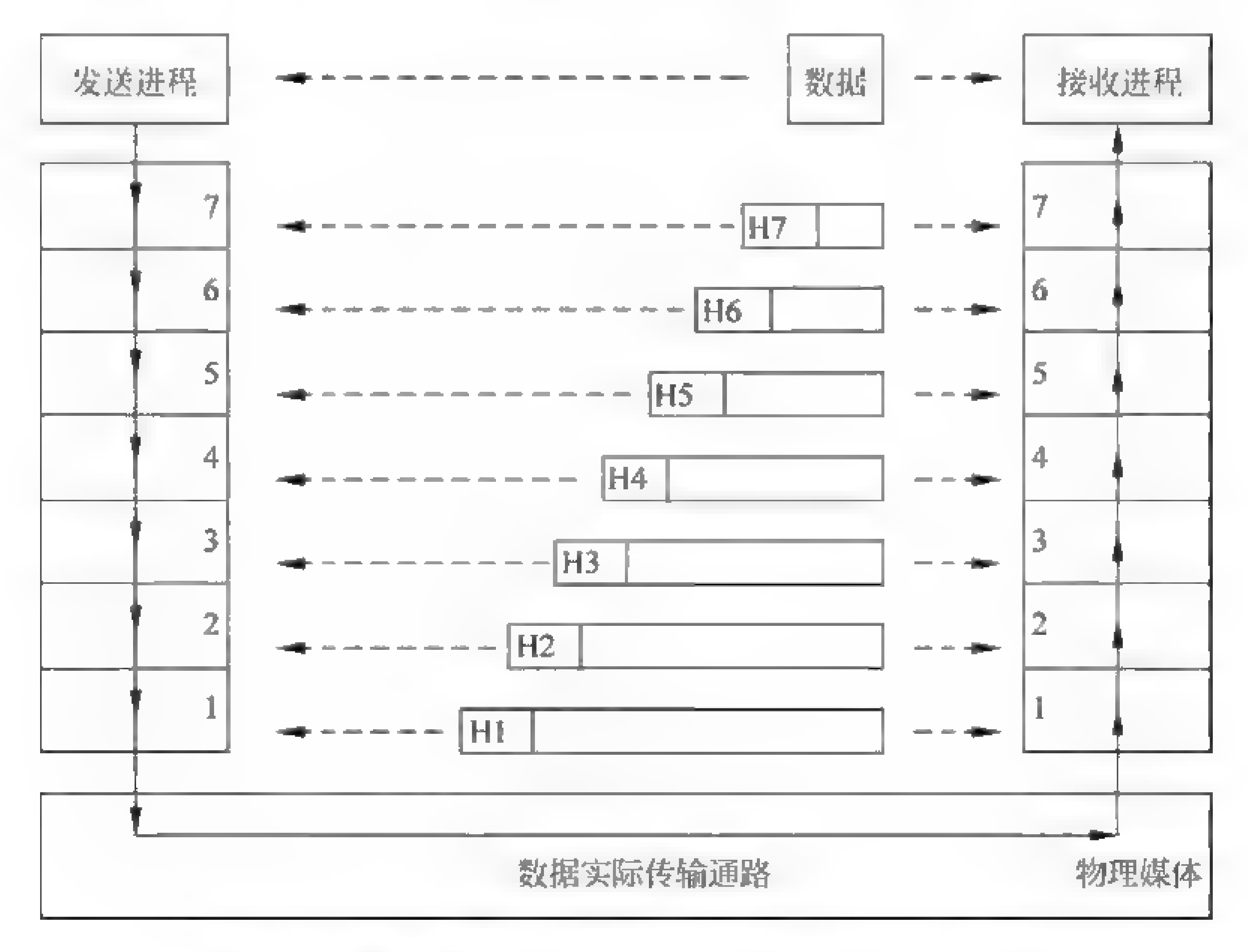


图1-3 OSI模型中数据在各层之间的传递过程

5) OSI 的安全体系结构

1982 年,OSI 基本模型建立之初,就开始进行 OSI 安全体系结构的研究。1989 年 12 月,ISO 颁布了计算机信息系统互连标准的第二部分,即 ISO 7498-2 标准,并首次确定了开放系统互连参考模型的安全体系结构,如图 1-4 所示。

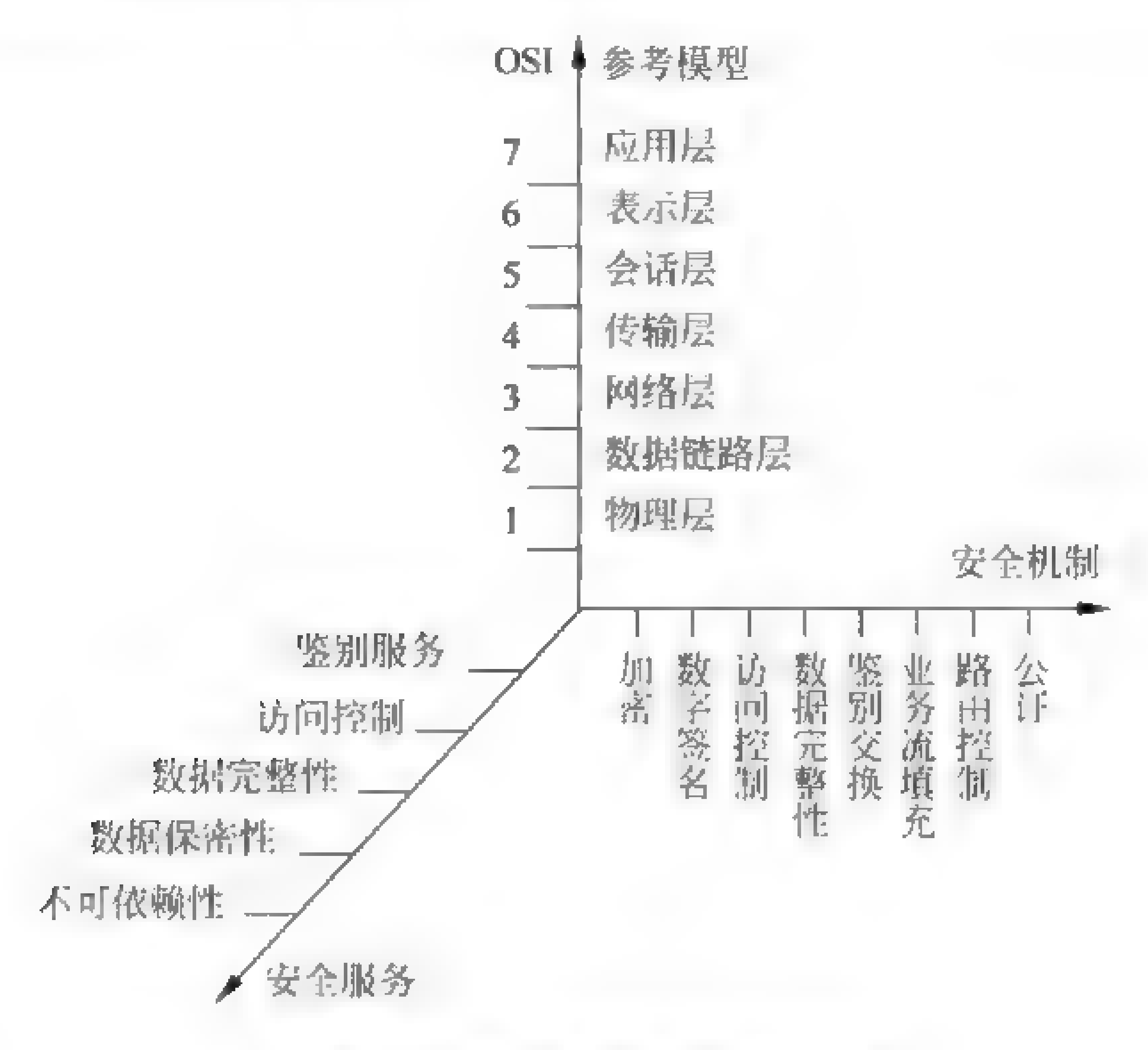


图 1-4 OSI 的安全体系结构

2. TCP/IP 模型

尽管 OSI 参考模型得到了全世界的认同,但是因特网历史上和技术上的开发标准都是传输控制协议/网际协议模型(Transmission Control Protocol/Internet Protocol,TCP/IP)。

TCP/IP 的标准是在名请求评议(Requests for Comments,RFC)的系列文档中发布的。RFC 描述因特网的内部运行。TCP/IP 标准总是以 RFC 的形式发布,但并非所有 RFC 都是标准的。一些 RFC 只提供情报信息、实验信息或历史信息。RFC 最初以因特网草案的形式拟定;它们通常由 IETF 职能小组中的一个或多个创作者开发。IETF 职能小组是由一些在 TCP/IP 套件的某一技术领域中具有特定职责的个人所组成的团队。IETF 将以 RFC 的形式发布因特网草案的最终版本,并为其分配一个 RFC 编号。

从协议分层模型方面来讲,TCP/IP 并不完全符合 OSI 的 7 层参考模型。TCP/IP 由 4 个层次组成:网络接口层、网络层、传输层、应用层。而 TCP/IP 通信协议采用了 4 层的层级结构,每一层都呼叫它的下一层所提供的网络来完成自己的需求。图 1-5 给出了 OSI 模型与 TCP/IP 模型的对照关系。

1) 应用层

应用层对应于 OSI 参考模型的高层,为用户提供所需要的各种服务,例如,FTP、Telnet、DNS、SMTP、POP3 等。

2) 传输层

传输层对应于 OSI 参考模型的传输层,为应用层实体提供端到端的通信功能。其功能包括:①格式化信息流;②提供可靠传输。为实现后者,传输层协议规定接收端必须发回

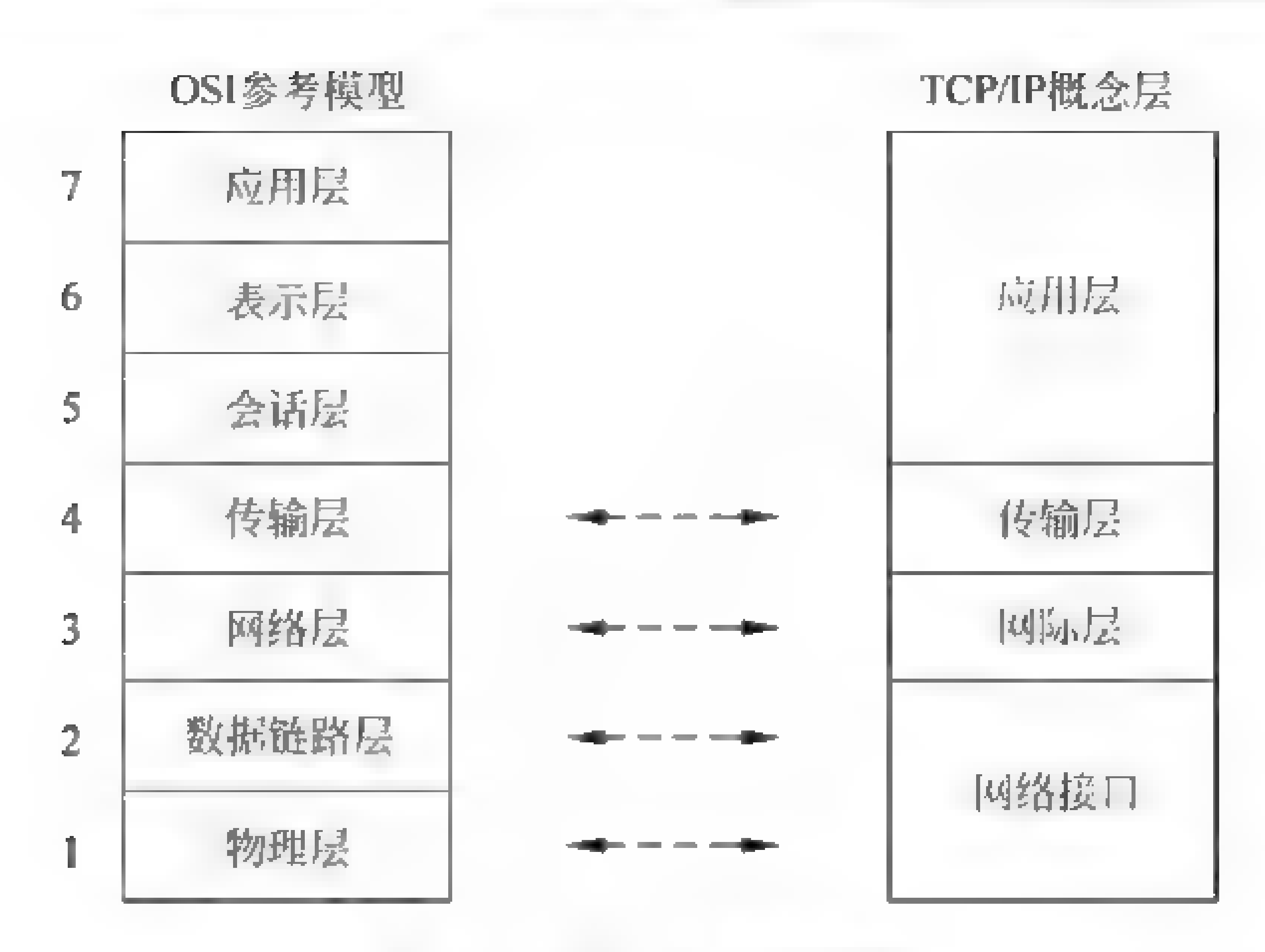


图1-5 TCP/IP参考模型

确认，并且假如分组丢失，必须重新发送。该层定义了两个主要的协议：传输控制协议(Transfer Control Protocol，TCP)和用户数据报协议(User Datagram Protocol，UDP)。TCP提供的是一种可靠的、面向连接的数据传输服务；而UDP提供的是不可靠的、无连接的数据传输服务。

3) 网际互联层

网际互联层对应于OSI参考模型的网络层，主要解决主机到主机的通信问题，负责相邻计算机之间的通信。其功能包括三方面：①处理来自传输层的分组发送请求，收到请求后，将分组装入IP数据报，填充报头，选择去往信宿机的路径，然后将数据报发往适当的网络接口。②处理输入数据报：首先检查其合法性，然后进行寻径——假如该数据报已到达信宿机，则去掉报头，将剩下部分交给适当的传输协议；假如该数据报尚未到达信宿，则转发该数据报。③处理路径、流控、拥塞等问题。

网际互联协议层包括：网际协议(Internet Protocol，IP)、控制报文协议(Internet Control Message Protocol，ICMP)、地址转换协议(Address Resolution Protocol，ARP)、反向地址转换协议(Reverse ARP，RARP)。IP是网络层的核心，通过路由选择将下一跳IP封装后交给接口层。IP数据报是无连接服务。ICMP是网络层的补充，可以回送报文。用来检测网络是否通畅。ARP是正向地址解析协议，通过已知的IP，寻找对应主机的MAC地址。RARP是反向地址解析协议，通过MAC地址确定IP地址，比如无盘工作站和DHCP服务。

4) 网络接口层

网络接口层与OSI参考模型中的物理层和数据链路层相对应。物理层是定义物理介质的各种特性：①机械特性；②电子特性；③功能特性；④规程特性。数据链路层是负责接收IP数据报并通过网络发送之，或者从网络上接收物理帧，抽出IP数据报，交给IP层。

常见的网络接口层协议有：Ethernet 802.3、Token Ring 802.5、X.25、Frame Reley、HDLC、PPP ATM等。

1.4.2 操作系统

1. Windows操作系统

Microsoft Windows是美国微软公司研发的一套操作系统，它问世于1985年，起初仅

仅是 Microsoft-DOS 模拟环境，后续的系统版本由于微软不断的更新升级，不但易用，也慢慢成为人们最喜爱的操作系统。

Windows 采用了图形化模式 GUI，比起从前的 DOS 需要输入指令使用的方式更为人性化。随着计算机硬件和软件的不断升级，微软的 Windows 也在不断升级，从架构的 16 位、32 位再到 64 位，系统版本从最初的 Windows 1.0 到人们熟知的 Windows 95、Windows 98、Windows ME、Windows 2000、Windows 2003、Windows XP、Windows Vista、Windows 7、Windows 8、Windows 8.1、Windows 10 和 Windows Server 服务器企业级操作系统，不断持续更新，微软一直在致力于 Windows 操作系统的开发和完善。

2. Linux 操作系统

Linux 是一套免费使用和自由传播的类 UNIX 操作系统，是一个基于 POSIX 和 UNIX 的多用户、多任务、支持多线程和多 CPU 的操作系统。它能运行主要的 UNIX 工具软件、应用程序和网络协议。它支持 32 位和 64 位硬件。Linux 继承了 UNIX 以网络为核心的设计思想，是一个性能稳定的多用户网络操作系统。

Linux 操作系统诞生于 1991 年 10 月 5 日(这是第一次正式向外公布时间)。Linux 存在着许多不同的版本，但它们都使用了 Linux 内核。Linux 可安装在各种计算机硬件设备中，比如手机、平板电脑、路由器、视频游戏控制台、台式计算机、大型计算机和超级计算机中。

严格来讲，Linux 这个词本身只表示 Linux 内核，但实际上人们已经习惯了用 Linux 来形容整个基于 Linux 内核，并且使用 GNU 工程各种工具和数据库的操作系统。

Linux 的基本思想有两点：第一，一切都是文件；第二，每个软件都有确定的用途。其中第一条详细来讲就是系统中的所有都归结为一个文件，包括命令、硬件和软件设备、操作系统、进程等对于操作系统内核而言，都被视为拥有各自特性或类型的文件。至于说 Linux 是基于 UNIX 的，很大程度上也是因为这两者的基本思想十分相近。

1) 完全免费

Linux 是一款免费的操作系统，用户可以通过网络或其他途径免费获得，并可以任意修改其源代码。这是其他的操作系统所做不到的。正是由于这一点，来自全世界的无数程序员参与了 Linux 的修改、编写工作，程序员可以根据自己的兴趣和灵感对其进行改变，这让 Linux 吸收了无数程序员的精华，不断壮大。

2) 完全兼容 POSIX 1.0 标准

这使得可以在 Linux 下通过相应的模拟器运行常见的 DOS、Windows 的程序。这为用户从 Windows 转到 Linux 奠定了基础。许多用户在考虑使用 Linux 时，就想到以前在 Windows 下常见的程序是否能正常运行，这一点就消除了他们的疑虑。

3) 多用户、多任务

Linux 支持多用户，各个用户对于自己的文件设备有自己特殊的权利，保证了各用户之间互不影响。多任务则是现在计算机最主要的一个特点，Linux 可以使多个程序同时并独立地运行。

4) 良好的界面

Linux 同时具有字符界面和图形界面。在字符界面用户可以通过键盘输入相应的指令来进行操作。它同时也提供了类似 Windows 图形界面的 X-Window 系统，用户可以使用鼠

标对其进行操作。在 X-Window 环境中就和在 Windows 中相似，可以说是一个 Linux 版的 Windows。

5）支持多种平台

Linux 可以运行在多种硬件平台上，如具有 x86、680x0、SPARC、Alpha 等处理器的平台。此外，Linux 还是一种嵌入式操作系统，可以运行在掌上电脑、机顶盒或游戏机上。2001 年 1 月发布的 Linux 2.4 版内核已经能够完全支持 Intel 64 位芯片架构。同时 Linux 也支持多处理器技术。多个处理器同时工作，使系统性能大大提高。

3. Android 操作系统

2007 年 11 月，Google 发布了基于 Linux 内核的开源智能移动操作系统 Android，该系统拥有庞大的用户数量和应用市场。来自 Gartner 的统计数据显示，2013 年第三季度全球智能手机的销售量为 1.5 亿多台，其中 Android 系统占据了 81.9%；而截至 2014 年 1 月 8 日仅 Android 官方应用市场 Google Play 上的应用数量就达到了 103 万。

尽管 Android 设备的用户数量庞大，但用户的安全意识却有待增强。来自美国《消费者报告》的年度 *State of the Net* 报告中则指出，在美国近 40%的手机用户没有采取适当的安全措施，2012 年有 560 万人遭遇过账户未经准许被擅自访问等问题。同时，虽然 Android 的应用数量巨大，但其应用安全性令人堪忧。TrustGo 公司的分析应用报告显示，Google Paly 上 3.15%的应用可能泄漏用户隐私或者存在恶意行为，而在国内知名的 91 应用市场上该比例为 19.7%。我国用户无法直接从 Google Play 上下载应用，导致了大量的、管理混乱的第三方应用市场的存在，对于 Android 设备的安全造成了严重的威胁。目前，越来越多的研究者开始关注 Android 上的安全问题和解决方案。

Android 与 Linux 的主要区别如下。

（1）从体系结构上看，Android 虽然运行在 Linux 内核之上，但不同于 GNU/Linux，Android 以 Bionic 取代 Glibc；以 Skia 取代 Cairo；以 Opencore 取代 Ffmpeg，等等。

（2）Android 的 Linux 内核实现了包括安全、存储器管理、程序管理、网络堆栈、驱动程序模型在内的模块。从进程间通信机制看，Android 增加了一种进程间的通信机制，即 IPC Binder。在内核源代码中，驱动程序文件为 coredroid/include/linux/binder.h 和 coredroid/drivers/android/binder.c。Binder 通过守护进程 Service Manager 管理系统中的服务，负责进程间的数据交换。各进程通过 Binder 访问同一块共享内存，以实现数据通信。

（3）从应用层的角度看，进程通过访问数据守护进程获取用于数据交换的程序框架接口，调用并通过接口共享数据。其他进程要访问数据，只需与程序框架接口进行交互，方便了程序员开发需要交互数据的应用程序。

（4）从内存管理看，在内存管理模块中，Android 内核采用了一种不用于标准 Linux 内核的低内存管理策略。在标准 Linux 内核当中，使用一种叫做内存泄露（Out of Memory，OOM）的低内存管理策略，即当内存不足时，系统检查所有的进程，并对进程进行限制评分，获得最高分的进程将被关闭（内核进程除外）。Android 系统采用的则是一种叫做 LMK（Low Memory Killer）的机制，这种机制将进程按照重要性进行分级、分组。内存不足时，将处于最低级别组的进程关闭。例如，在移动设备当中，UI 界面处于最高级别，所以该进程永远不会被中止，这样，在终端用户看来，系统是稳定运行的。与此同时，Android 新增加了一种内存共享的处理方式——匿名共享内存（Anonymous Shared Memory，Ashmem）。通过

Ashmem，进程间可以匿名自由共享同名的内存块；这种共享方式在标准 Linux 当中不被支持。

每个操作系统平台都有自己的特点，而且大部分功能都有重叠部分，只是在某些实现上有所不同。例如 TCP、IP 协议栈无论在 Linux 和 Windows 下都有实现，但是它们的实现方法有所区别，功能有所不同。还有网络编程接口 Socket，在 Linux 下的实现和 Windows 下有不同之处，并且即使在 Windows 平台，各种 Windows 版本的实现都有所不同。所以，在进行网络安全开发的时候，要了解所基于的开发平台。如果在 Linux 下进行开发，那么对于 Linux 系统下的网络部分要有所了解，包括它的实现机理，最好是通过阅读其源代码来进行深入理解，对于 Windows 系统，由于其源代码不公开，那么必须对其底层的网络机制有所了解，深入理解 Windows 系统编程技术，了解不同版本的区别，有针对性地进行开发。

1.4.3 网络安全组成

网络安全是一个非常全面而复杂的课题，它包含的内容极其庞大，在这里把网络分成三个组成部分，分别是客户端、服务器和网络设施。从这个方面来考虑网络安全的问题。

1. 客户端安全

在客户端部分主要涉及具体用户使用网络的安全问题。例如，个人数据的安全性。用户把计算机连入网络后，其就是一个客户端。要保证个人数据的安全，需要建立良好的使用网络的习惯。例如，定时杀毒，安装个人防火墙，不打开可疑邮件，不随便安装来路不能明的软件。对个人敏感的信息要进行加密处理，放在一个安全的地方，对重要的数据要进行备份，以免硬件损坏而丢失数据。要及时升级操作系统，安装各种补丁程序，或者下载新的软件应对安全问题。在客户端的一些常用软件最易受到安全威胁。还有一些比较常用的软件，例如文字处理软件、媒体播放器软件、电子邮件接收和阅读软件，也容易受到威胁。

2. 服务器安全

服务器方面的安全性也至关重要，很多重要的数据都放在服务器里面。对于重要数据的保护则需要更加强大的安全措施，例如，使用专业的防火墙设备、入侵检测系统、安全扫描系统，针对特定服务器的保护系统。

在应用服务器的服务软件时，软件的合理配置也很重要，对于不需要开放的服务尽量不要开放。对于开放的服务要有足够的安全措施，例如，设置安全权限以及密钥。

最常用的客户端和服务器系统是 Web 浏览器和 Web 服务器系统，它们的安全性在网络安全内容中占据重要的地位，针对 Web 浏览器和 Web 服务器的攻击数量庞大，种类繁多，危害性也很大。针对 Web 安全提出了很多技术，例如，安全套接层（Secure Sockets Layer，SSL）、TLS、SHTTP 等。SSL 使用 TCP 提供一个可靠的端到端安全服务，为两个通信个体之间提供保密性和完整性。SSL 包括 SSL 握手协议和 SSL 记录协议两个部分：SSL 记录协议建立在可靠的传输协议基础上，提供链接的保密性和完整性，用来封装高层的协议；SSL 握手协议完成客户端和服务器端的安全鉴别、协商加密算法和密钥，提供身份鉴别和可靠协商的功能。

3. 网络设施安全

网络设施的安全保护重要的网络设备，例如路由器交换机等重要设备，防止由于故障例

如停电以及其他自然灾害造成系统瘫痪，电源故障会造成设备断电，导致操作系统引导失败或数据信息丢失等。

1.4.4　网络安全开发包

网络安全性问题关系到未来网络应用的深入发展，它涉及安全策略、移动代码、指令保护、密码学、操作系统、软件工程和网络安全管理等内容。主要的网络技术包括：密码技术、网络安全协议、防火墙技术、入侵检测、恶意代码和病毒防护以及密钥管理等。

网络安全本身理论性和实践性都很强，掌握了网络安全相关的基本概念、基本原理后可以运用到实际的工程中，在实际应用中，需要针对实际应用环境开发一些特定应用程序以提供相应的安全服务，而这时掌握一些实用的网络安全编程工具就显得尤为重要。

什么是网络安全开发包？网络安全开发包是指用于网络安全研究和开发的一些专用编程接口或开发包，它的主要作用是提供网络安全研究和开发的基本功能的实现，为研究者和开发者进一步研究和开发网络安全提供编程接口，为网络安全服务的实现提供方便。

一般一种网络安全开发包是针对特定的网络安全服务而开发的，它可以提供某一种或多种网络安全服务。网络安全开发包都经过很多网络安全研究和开发者的长期研究而成，通过不断测试和使用而逐渐成熟，并在实际的应用中得到推广。

某些网络安全开发包基本上已经实现了某个特定网络安全服务的基本框架和基本功能，然后开发者可以在这个已经构造好的基本框架下进行进一步的开发，这样就为开发者节省了时间和精力，为开发功能更强大的系统提供方便。

网络安全开发包的种类较多，其实现的功能也不尽相同，下面介绍一些常见的网络安全开发包。

1. CryptoAPI

作为 Microsoft Windows 的一部分提供的应用程序编程接口(Application Programming Interface，API)，CryptoAPI 提供了一组函数，包括编码、解码、加密、解密、哈希计算等，这些函数允许应用程序在对用户的敏感私钥数据提供保护时以灵活的方式对数据进行加密或数字签名。实际的加密操作是由称为加密服务提供程序的独立模块执行的。

2. OpenSSL

OpenSSL 整个软件包大概可以分成三个主要的功能部分：SSL 协议库、应用程序以及密码算法库。OpenSSL 的目录结构自然也是围绕这三个功能部分进行规划的。

作为一个基于密码学的安全开发包，OpenSSL 提供的功能相当强大和全面，囊括主要的密码算法、常用的密钥和证书封装管理功能以及 SSL 协议，并提供了丰富的应用程序供测试或其他目的使用。

上述两种网络安全开发包是重点，除此之外，在实际应用中还会使用以下的网络安全开发包。

3. Crypto++

Crypto++是一个用 C++编写的密码学类库，非常强大，在密码学界很受欢迎。Crypto++有其明显的优点，主要是功能全，统一性好。基本上密码学中需要的主要功能都可以在里面找得到。目前最新的版本是 Crypto++ Library 5.6，可以适应各种常用的操作系统和编译平

台。参考网站 http://www.cryptopp.com/。

4. 网络数据捕获开发包 Libpcap 和 WinPcap、WireShark

Libpcap 的英文意思是 Packet Capture library，即数据包捕获函数库。Libpcap 是 UNIX Linux 平台下的网络数据包捕获函数包，大多数网络监控软件都以它为基础。该库提供的 C 函数接口可用于需要捕获经过网络接口（只要经过该接口，目标地址不一定为本机）数据包的系统开发上。由 Berkeley 大学 Lawrence Berkeley National Laboratory 研究院的 Van Jacobson、Craig Leres 和 Steven McCanne 编写。该函数库支持 Linux、Solaris 和 * BSD 系统平台。WinPcap 是 Libpcap 在 Windows 下的版本，其官方网站是 http://winpcap.polito.it/。WireShark（前称 Ethereal）是一个网络封包分析软件。网络封包分析软件的功能是撷取网络封包，并尽可能显示出最为详细的网络封包资料。WireShark 使用 WinPcap 作为接口，直接与网卡进行数据报文交换。

5. 网络入侵开发包 Libnids

Libnids 是一个用于网络入侵检测开发的专业编程接口，它使用了 Libpcap，所以它具有捕获数据包的功能。同时，Libnids 提供了 TCP 数据流重组功能，所以对于分析基于 TCP 的各种协议 Libnids 都能胜任。Libnids 还提供了对 IP 分片进行重组的功能，以及端口扫描检测和异常数据包检测功能，可以快速实现网络入侵检测的基本功能。

6. 防火墙开发包 NetFilter

NetFilter 是由 Rusty Russell 提出的 Linux 2.4 内核防火墙框架，该框架既简洁又灵活，可实现安全策略应用中的许多功能，如数据包过滤、数据包处理、地址伪装、透明代理、动态网络地址转换（Network Address Translation，NAT），以及基于用户及媒体访问控制（Media Access Control，MAC）地址的过滤和基于状态的过滤、包速率限制等。

7. 安全电子邮件开发包 Sendmail

Sendmail 是 Linux UNIX 下的邮件服务器。Sendmail 作为一种免费的邮件服务器软件，已被广泛地应用于各种服务器中，它在稳定性、可移植性、及确保没有 Bug 等方面具有一定的特色，当前其最新的稳定版本为 8.14.3。

1.5 本书内容安排

本书旨在培养学生网络安全编程的基本思想，学习网络安全编程的主要方法和工具，后面几章的具体内容安排如下。

第 2 章阐述 Socket 编程以及 Visual C++网络安全编程的核心技术。

第 3 章、第 4 章根据密码学的经典算法，分别阐述密码学编程，基于网络安全开发包 OpenSSL 的安全编程。

第 5 章、第 6 章和第 7 章分别讲述网络扫描器的设计、防火墙和入侵检测系统设计。安全扫描技术与防火墙、安全监控系统互相配合能够提供很高安全性的网络。

第 8 章讲述应用系统安全编程，包括安全 Web 程序设计和安全电子邮件编程。

小　结

本章从技术层面和国家战略两个方面阐述了网络空间安全的必要性,介绍了网络空间安全学科的研究内容、分析了网络空间安全学科人才培养的要求。对网络安全程序设计所需的基础知识做了一个全面的介绍。

思 考 题

1. 阐述网络空间安全必要性。
2. 网络空间安全学科主要研究内容包括哪些?
3. 常见的网络安全开发包有哪些?
4. 比较:ISO 协议与 TCP/IP 协议的异同。

第 2 章　网络安全编程基础

网络安全具有很强的理论性和实践性，在掌握了网络安全相关的概念、基本原理之后，就需要将相关的理论运用到实际的工程中。套接字编程是网络安全编程的基础，Windows 平台上基于 WinSock 实现基本的套接字编程，在此基础之上，介绍基于 Visual C++的网络安全编程实例。

2.1　套接字编程

2.1.1　套接字概念

1. 概述

套接字(Socket)是由加利福尼亚大学伯克利分校开发的一个编程接口，目的是为了在 UNIX 操作系统下实现 TCP/IP，方便调用网络操作。它其实就是一套应用程序接口(Application Programming Interface，API)，此 API 称为 Socket 接口，现在 Socket 接口几乎是 TCP/IP 网络标准的 API，很多 TCP/IP 的网络应用程序都是基于 Socket 编写的。

Socket 是用来实现主机和主机通信的一个接口，通过它可以完成主机间的通信操作，它屏蔽了底层的协议细节，让用户能够实现各种类型的通信操作。它是应用程序对应的进程和网络协议之间的接口，位置如图 2-1 所示。套接字的出现，为网络应用程序的编写提供了极大的方便。

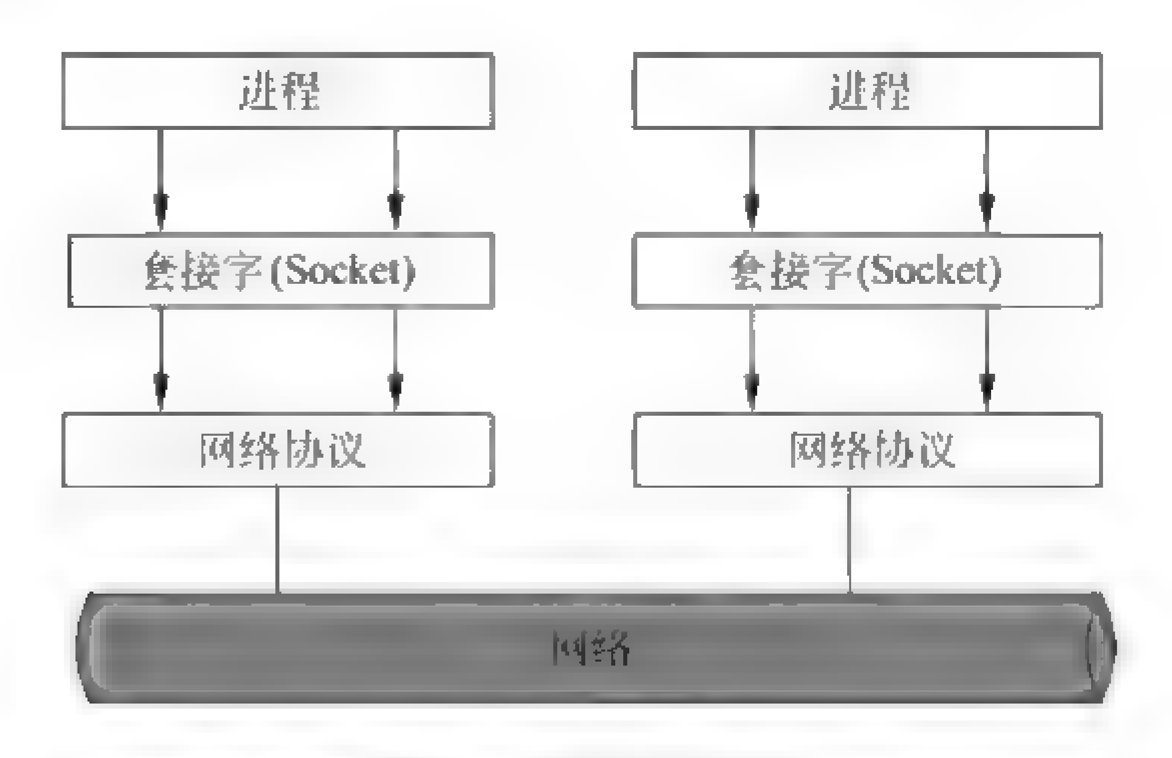


图 2-1　套接字的位置

如图 2-2 所示，套接字是一个双向通信设备，可用于同一台主机上不同进程之间的通信，也可用于沟通位于不同主机的进程。套接字是所有进程间通信方法中唯一允许跨主机通信的方式。很多因特网程序，如 Telnet、rlogin、FTP、talk 和万维网都是基于套接字的，用户可以用一个 Telnet 程序从一台网页服务器上获取一个万维网网页，因为它们都使用套接

字作为网络通信方式。可以通过执行 telnet www.codesourcery.com 80 连接到位于 www.codesourcery.com 主机的网页服务器。80 指明了连接的目标进程是运行于 www.codesourcery.com 的网页服务器，而不是其他进程。成功建立连接后，试着输入“GET /”，这会通过套接字发送一条消息给网页服务器，而相应的回答则是服务器将主页的 HTML 代码传回，然后关闭连接。具体代码如下。

```
telnet www.codesourcery.com 80
Trying 206.168.99.1...
Connected to merlin.codesourcery.com (206.168.99.1).
Escape character is '^]'.
GET /
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1">
```

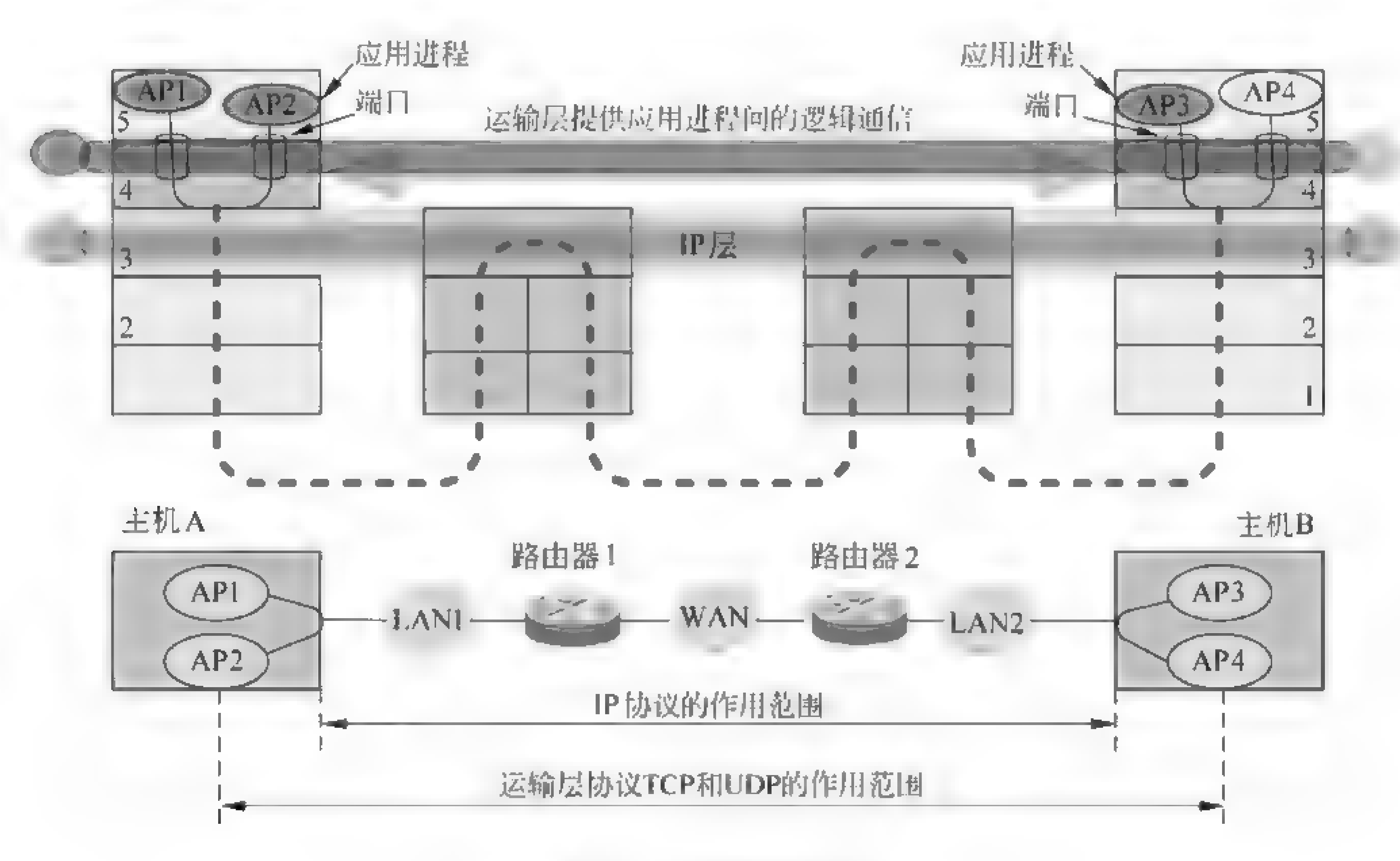


图 2-2　进程与套接字

2. 主要参数

不同应用程序进程间的网络通信和连接主要根据以下三个参数进行区分：通信的目的 IP 地址、使用的传输层协议（TCP 或 UDP）和使用的端口号。Socket 的原意是“插座”，将这三个参数结合起来，与一个“插座”Socket 进行绑定，应用层就可以通过套接字接口和传输层进行交流，区分来自不同应用程序进程或网络连接的通信，实现数据传输的并发服务。

在两个程序进行通信连接的过程中，Socket 可以被看作是一个端点，是连接应用程序和网络驱动程序的桥梁，在应用程序中创建对应的 Socket，通过绑定将其与网络驱动建立关系。此后，应用程序送给 Socket 的数据，由 Socket 交给网络驱动程序向网络发送出去。

计算机从网络上收到与该Socket绑定IP地址和端口号相关的数据后，由网络驱动程序交给Socket，应用程序便可从该Socket中提取接收到的数据，网络应用程序就是这样通过Socket进行数据的发送与接收的。

因此，IP地址和端口号的组合被称为套接字，其用于标识客户端请求的服务器和服务。为了后续内容的正确理解，接下来对以下概念进行详细说明。

1）端口

端口是一种抽象的软件结构，应用程序通过系统调用与某端口建立连接后，传输层给该端口的数据都被相应的进程接收，相应的进程发给传输层的数据都通过该端口输出。

端口号：一个整型标识符，用来表示端口，取值为0～65 535，1024以下的端口保留给预定义的服务。

需要注意的是，TCP/IP传输层的两个协议TCP与UDP是完全独立的两个软件模块，因此各自端口独立，也就是说TCP/UDP可以拥有相同的端口号。

2）IP地址

所谓IP地址，就是给每个连接在因特网上的主机分配一个独一无二的地址。按照TCP/IP协议规定，IP地址用二进制表示，每个IP地址长32b，比特换算成字节，就是4B。

3）套接字

套接字存在于通信域中，通信域也叫地址簇，它是一个抽象的概念，主要作用是将通过套接字通信的进程的共有特性综合在一起，套接字通常只与同一区域的套接字交换数据，当然，在执行某种转换进程后也可以实现跨区通信。Windows Sockets只支持一个通信域，即网际域（AF_INET），这个域被使用网际协议簇的通信进程所使用。

3. 套接字分类

常用的TCP/IP协议的套接字类型有如下3种。

1）流套接字（SOCK_STREAM）

流套接字用于提供面向连接、可靠的数据传输服务，该服务将保证数据能够实现无差错、无重复的发送，并按顺序进行接收。流套接字之所以能够实现可靠的数据服务，原因在于其使用了面向连接的传输控制协议，即TCP。

2）数据报套接字（SOCK_DGRAM）

数据报套接字提供了一种无连接的服务，该服务并不能保证数据传输的可靠性，数据有可能在传输过程中出现丢失或重复，且无法保证数据的按序到达。数据报套接字使用了无连接的UDP进行数据的传输，由于数据报套接字不能保证数据传输的可靠性，因此，对于有可能出现的数据丢失等情况，需要在程序中做相应的处理。

3）原始套接字（SOCK_RAW）

原始套接字（SOCKET_RAW）允许对较低层次的协议直接访问，比如IP、ICMP，它常用于检验新的协议实现或者访问现有服务中配置的新设备，因为原始套接字可以自如地控制Windows下的多种协议，能够对网络底层的传输机制进行控制，所以可以用原始套接字来操纵网络层和传输层应用。比如，可以通过原始套接字来接收发向本机的ICMP、IGMP协议包，或者接收TCP/IP栈不能处理的IP包，也可以用来发送一些自定包头或自定协议的IP包。网络监听技术很大程度上依赖于SOCKET_RAW。

原始套接字与标准套接字（指的是前面介绍的流套接字和数据报套接字）的区别在于：

原始套接字可以读写内核没有处理的 IP 数据包，而流套接字只能读取 TCP 的数据，数据报套接字只能读取 UDP 的数据。因此，如果要访问其他协议发送的数据，则必须使用原始套接字。

2.1.2　连接过程

在 TCP/IP 网络中，两个进程间相互通信所采用的主机模式是客户/服务器(Client/Server，CS)模式。该模式的建立基于以下两点：①非对等作用；②通信完全是异步的。客户/服务器模式在操作过程中采取的是主动请求方式。

首先服务器方要先启动，并根据请求提供相应的服务(过程如下)。

(1) 打开某通信通道并告知本地主机，它愿意在某一个公认地址上接收客户的请求。

(2) 等待客户请求到达该端口。

(3) 接收到重复服务请求，处理该请求并发送应答信号。

(4) 返回第(2)步，等待另一客户请求。

(5) 关闭服务器。

客户方：

(1) 打开通信通道，并连接到服务器所在主机的特定端口。

(2) 向服务器发送服务请求报文，等待并接收应答；继续提出请求……

(3) 请求结束后，关闭通信通道并终止。

根据连接启动的方式以及本地套接字要连接的目标，套接字之间的连接过程可以分为三个步骤：服务器监听，客户端请求，连接确认，如图 2-3 所示。

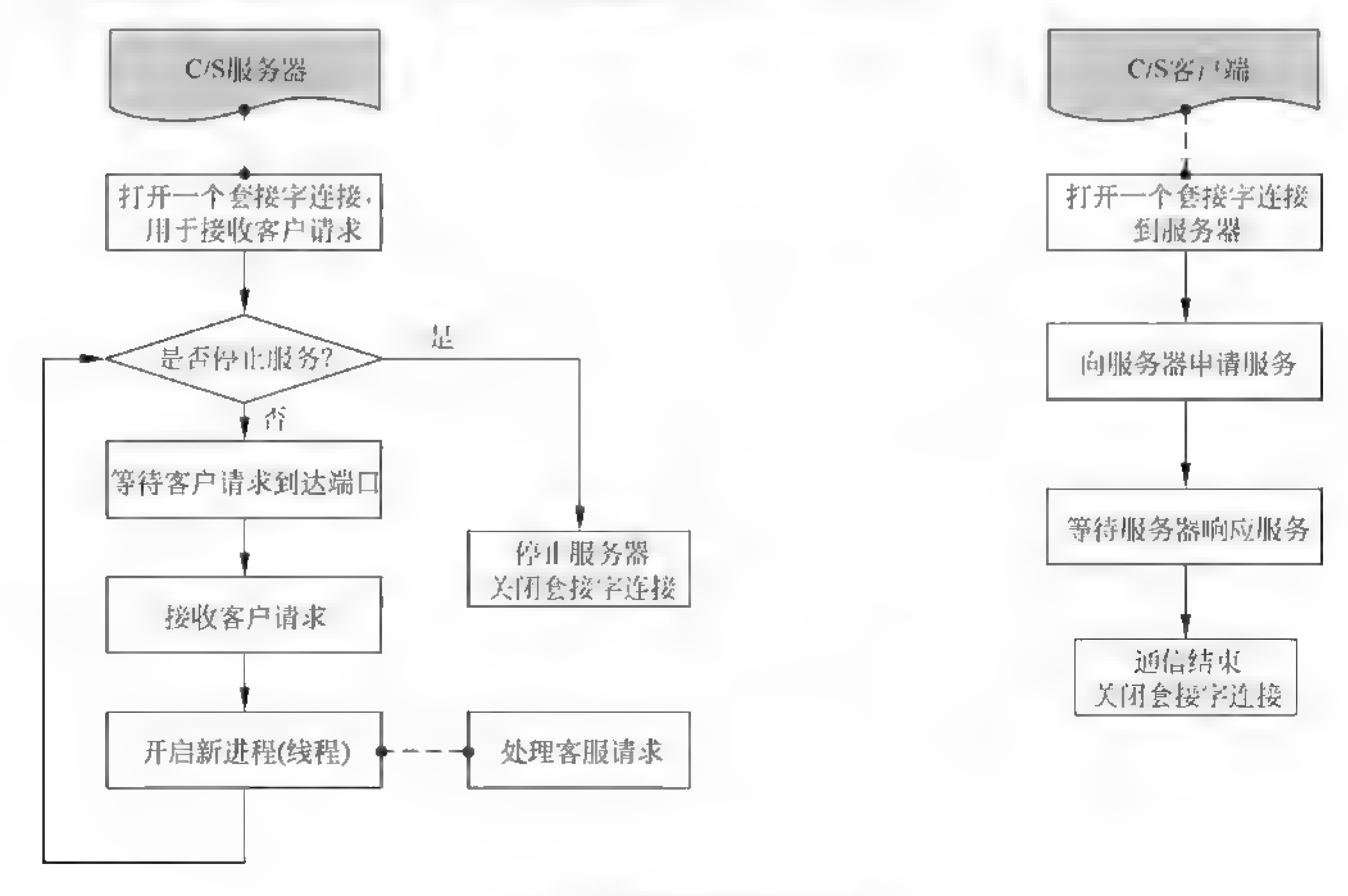


图 2-3　套接字通信示意图

(1) 服务器监听：服务器端套接字并不定位具体的客户端套接字，而是处于等待连接的状态，实时监控网络状态。

(2) 客户端请求：由客户端的套接字提出连接请求，要连接的目标是服务器端的套接字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务器端套接字的地址和端口号，然后向服务器端套接字提出连接请求。

(3) 连接确认：当服务器端套接字监听到或接收到客户端套接字的连接请求，它就响应客户端套接字的请求，建立一个新的线程，把服务器端套接字的描述发给客户端，一旦客户端确认了此描述，连接就建立好了。而服务器端套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。

套接字通信过程如图 2-3 所示。

2.1.3 基本套接字

一般来说，要进行网络通信，必须要在网络的每一端都建立一个套接字，任意两个套接字之间是可以建立连接的，也可以是无连接的，并通过对套接字的"读""写"操作实现网络通信功能，类似于文件的打开、读、写、关闭的方式。

套接字有以下三种类型。

(1) 数据流套接字(SOCK_STREAM)：对应 TCP。

(2) 数据报套接字(SOCK_DGRAM)：对应 UDP。

(3) 原始套接字(SOCK_RAW)：通过使用原始套接字，可以将网卡设为混杂模式，并且可以使捕获到的数据包不仅是单纯的数据信息，而是包含 IP 头、TCP 头等信息头的最原始的数据信息，这些信息保留了它在网络传输时的原貌，通过对这些在底层传输的原始信息的分析，可以得到更多的网络信息。

为了更好地说明套接字编程的原理，先介绍以下几个基本的套接字操作，其余的会在后续篇幅中给出更详细的使用说明。

1. 创建套接字——socket()

```
SOCKET PASCAL FAR socket(int af, int type,int protocol);
```

功能：创建一个新的套接字。

参数说明如下。

af：通信发生的区域，即协议的地址族，一般来说，IPv4 对应的是 AF_INET，这也是使用最多的协议。

type：要建立的套接字类型，若是面向连接的服务，则使用流套接字类型，值设为 SOCK_STREAM；若是无连接的服务，则使用数据报类型，值设为 SOCK_DGRAM。

protocol：使用的特定协议，对于面向连接的服务，可以设为 IPPROTO_TCP，对于面向无连接的服务，则可以设为 IPPROTO_UDP。

2. 指定本地地址——bind()

```
int PASCAL FAR bind(SOCKET s,const struct sockaddr FAR * name,int namelen);
```

功能：将套接字地址与所创建的套接字号联系起来，没有错误的情况下，bind()返回 0，

否则返回相应的错误值 SOCKET_ERROR。

参数说明如下。

s：由 socket()调用返回的并且未做连接的套接字描述符(套接字号)。

地址结构说明：

```
struct sockaddr_in
{
 short sin_family;
 u_short sin_port;
 struct in_addr sin_addr;
 char sin_zero[8];
}
```

参数说明如下。

sin_family：一般为 AF_INET。

sin_port：16 位端口号，网络字节顺序。

in_addr sin_addr：32 位 IP 地址，网络字节顺序。

sin_zero：保留。

3. 建立套接字连接——connect()和 accept()

```
int PASCAL FAR connect(SOCKET s,const struct sockaddr FAR * name,int namelen);
SOCKET PASCAL FAR accept(SOCKET s,struct sockaddr FAR * name,int FAR * addrlen);
```

功能：共同完成连接工作。

参数同上。

4. 监听连接——listen()

```
int PASCAL FAR listen(SOCKET s, int backlog);
```

功能：用于面向连接的服务器，表明它愿意接收连接。

参数说明如下。

backlog：指定被搁置的连接队列的最大长度，当连接的客户数大于该最大长度并且服务进程没来得及处理，则多出的连接请求会失败，目前允许的最大值为 5。

5. 数据传输——send()与 recv()

```
int PASCAL FAR send(SOCKET s,const char FAR * buf,int len,int flags);
int PASCAL FAR recv(SOCKET s,const char FAR * buf,int len,int flags);
```

功能：数据的发送与接收。

参数说明如下。

buf：指向存有传输数据的缓冲区的指针。

6. 多路复用——select()

```
int PASCAL FAR select(int nfds,fd_set FAR * readfds,fd_set FAR * writefds,
fd_set FAR * exceptfds,const struct timeval FAR * timeout);
```

功能：用来检测一个或多个套接字状态。

参数说明如下。

readfds：指向要做读检测的指针。

writefds：指向要做写检测的指针。

exceptfds：指向要检测是否出错的指针。

timeout：最大等待时间。

7. 关闭套接字——closesocket()

```
BOOL PASCAL FAR closesocket(SOCKET s);
```

功能：关闭套接字 s。

2.1.4 典型过程图

面向连接的套接字系统调用时序、面向无连接协议的套接字调用时序以及面向连接的应用程序分别如图 2-4～图 2-6 所示。

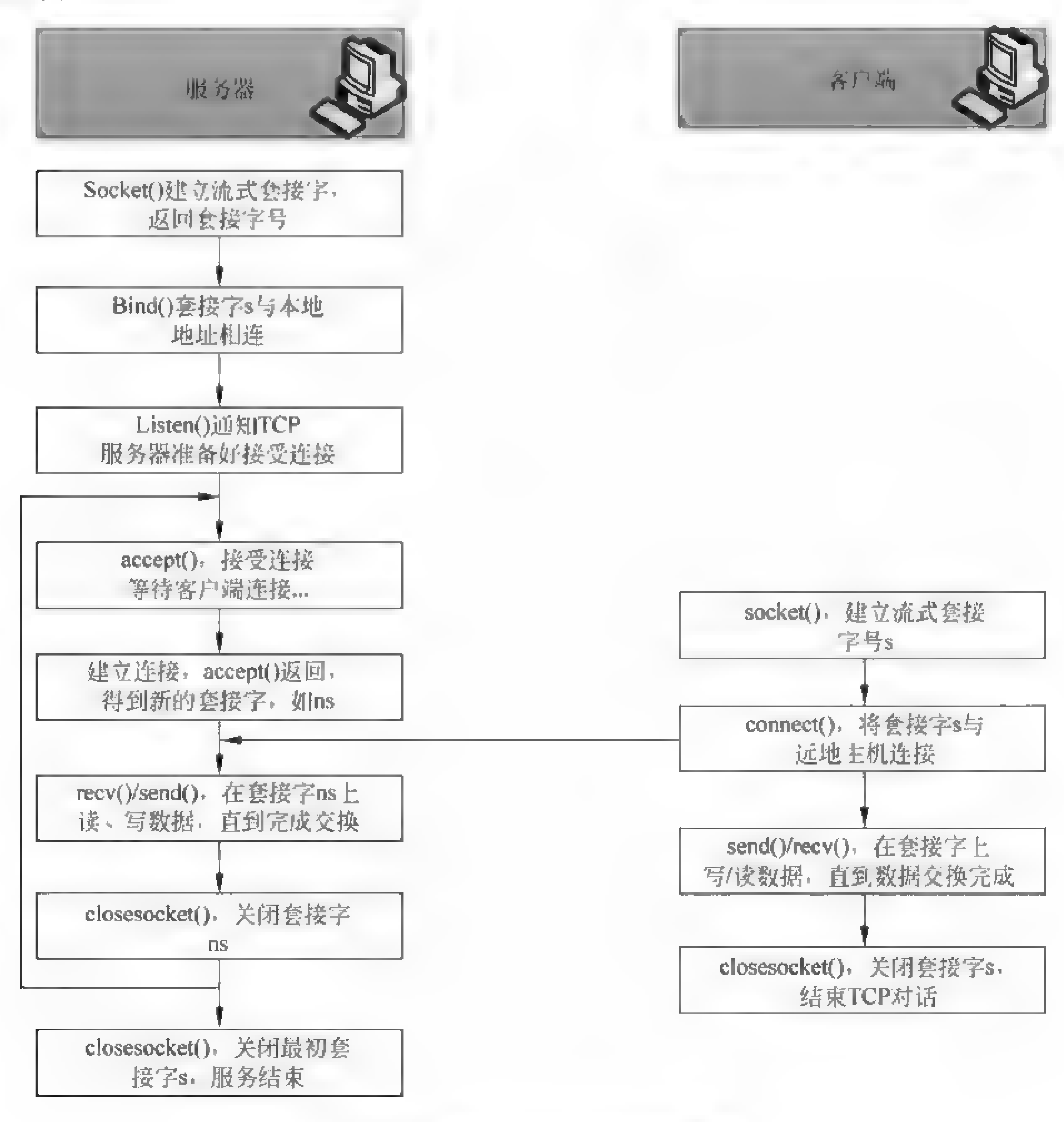


图 2-4 面向连接的套接字时序图

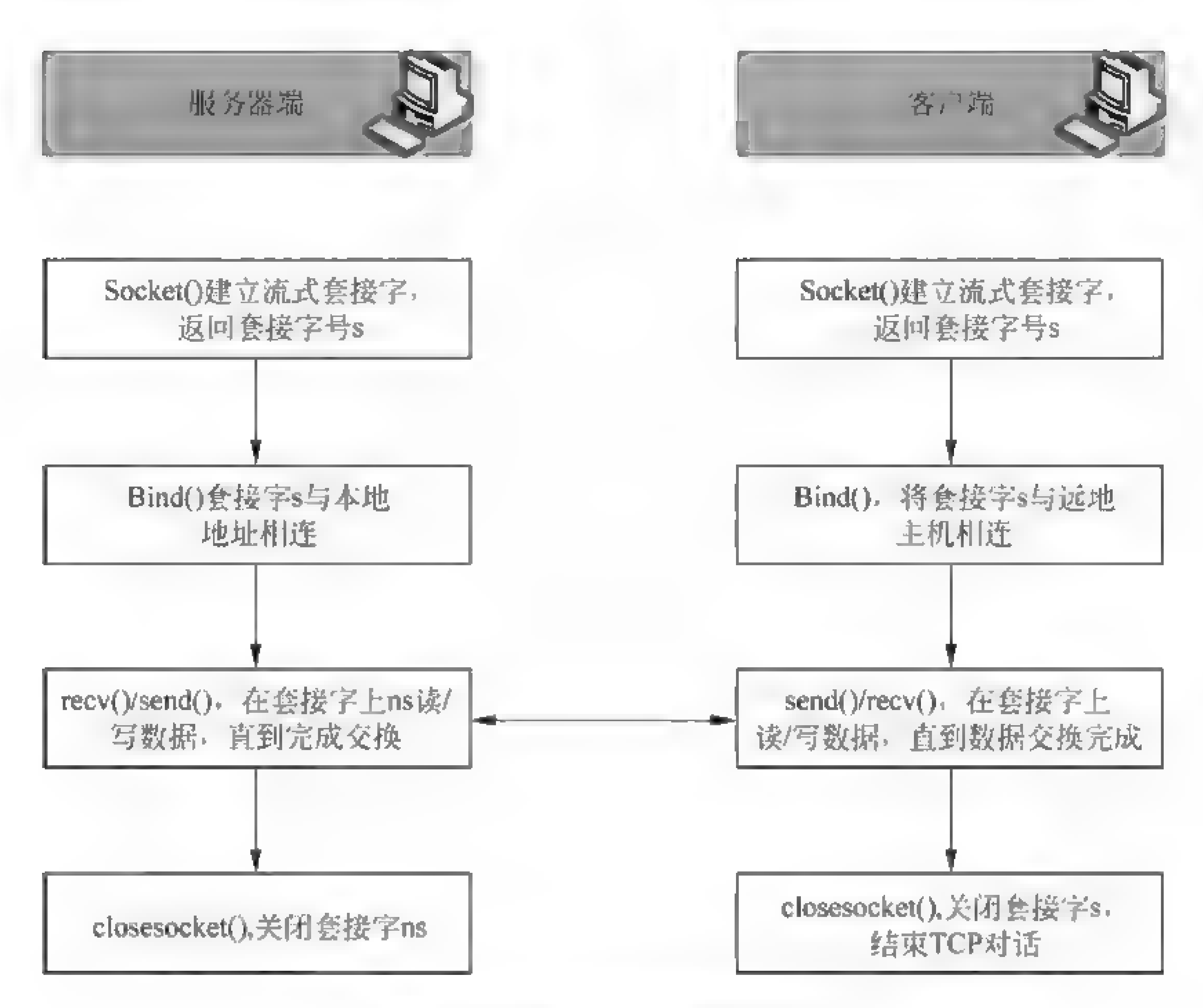


图 2-5　无连接套接字时序图

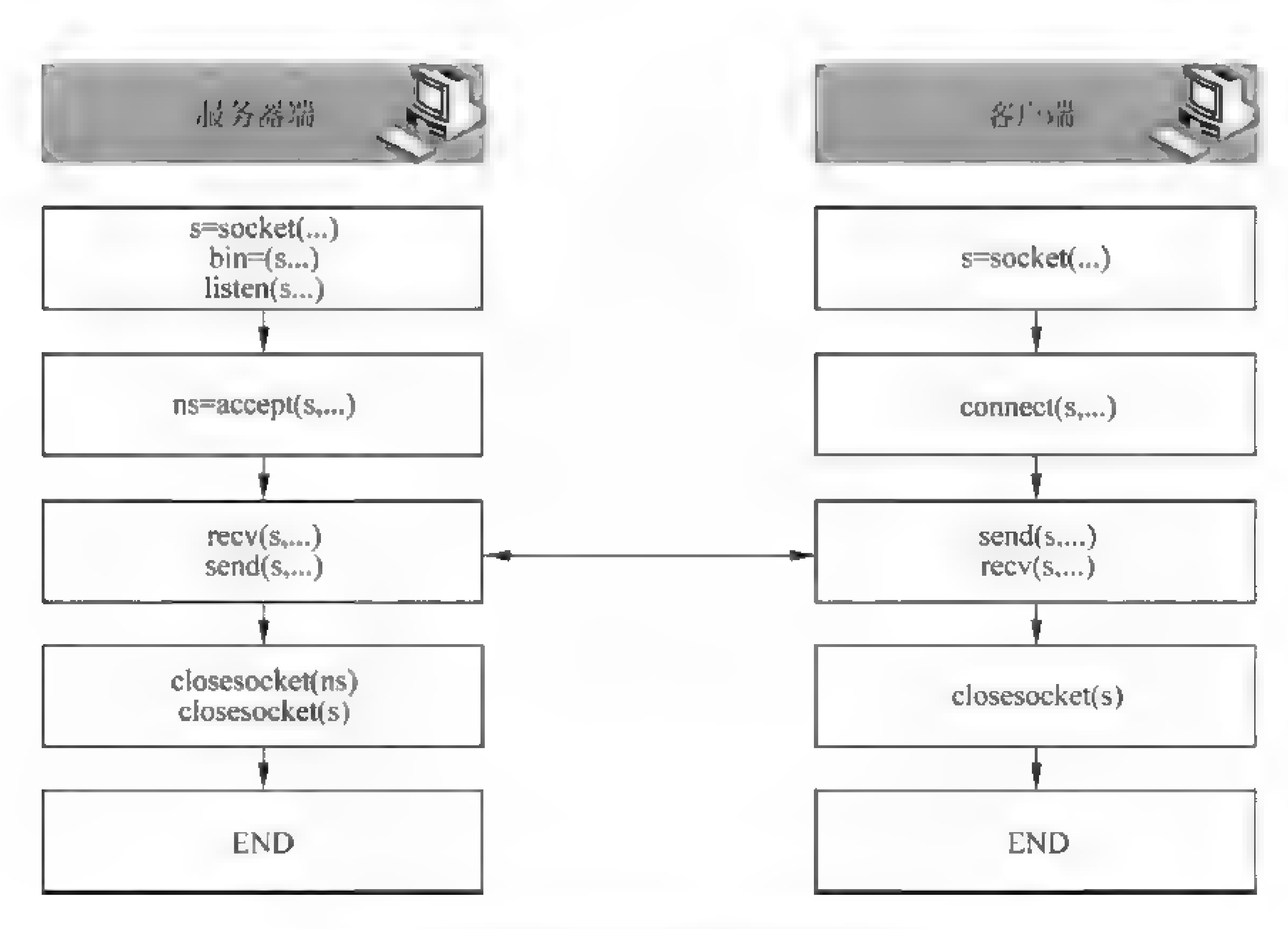


图 2-6　面向连接的应用程序流程图

2.2 WinSock编程相关函数

Windows Sockets API是Windows下用于网络通信的网络应用程序接口，通过前缀WSA区分，API函数有1.1版本和2.0版本，分别称为WinSock1和WinSock2，二者最主要的区别是1.1版本只支持TCP/IP协议，而2.0版支持多种协议。所有使用1.1版的源代码、二进制文件和应用程序都可以不加修改地在2.0规范下使用。为了适用于Windows下的消息机制和异步的I/O选择操作，Windows Sockets API在功能上扩充了将近二十个函数，其中，扩充的部分均冠以前缀WSA，充分体现了Windows的优越性。

为了适应近年来网络技术特别是多媒体网络技术的迅猛发展，WinSock套接字目前基本都是用版本2进行开发，通过制定Windows Sockets 2规范来提供一个与协议无关的网络传输接口，Windows Sockets 2实际上是Windows Sockets 1.1接口的一个超集，它在保持和Windows Sockets 1.1完全向后兼容能力的同时，扩展了Windows Sockets接口。

在使用WinSock编写程序时，需要用到几个重要文件，分别是头文件winsock2.h，静态链接库文件ws2_32.lib，以及动态链接库文件ws2_32.dll。头文件winsock2.h是用在源程序中的，静态链接库文件ws2_32.lib是用来编译基于WinSock的程序，而动态链接库文件ws2_32.dll是运行基于WinSock的程序所需要的。

2.2.1 Win32 API相关套接字常用函数

接下来对WinSock编程中的常用函数及使用方法进行介绍。

1. 套接字版本协商

函数原型：

```
int WSAStartup (
WORD wVersionRequested,
LPWSADATA lpWSAData
);
```

返回值：

如果ws_32.dll或底层网络子系统没有正确初始化或被找到，函数返回WSASYSNOTREADY。

如果请求的版本低于WinSock动态库所支持的最低版本，WSAStartup函数将返回WSAVERNOTSUPPORTED。

如果请求的版本等于或者高于WinSock动态库所支持的最低版本，WSAData的wVersion成员包含应用程序应该使用的版本，它是动态库所支持的最高版本与请求版本中较小的那个。

参数说明如下。

wVersionRequested：用来指定准备加载的WinSock库的版本。高字节位指定所需要的WinSock库的副版本，而低字节则是主版本。通常如果版本为2.1，则其中2为主版本号，1为副版本号。

lpWSAData：这是一个返回值，指向 WSADATA 结构的指针，WSAStartup 函数用其来加载的库版本有关的信息填在这个结构中。

2. 创建套接字

函数原型：

```
SOCKET socket (
int af,
int type,
int protocol
);
```

返回值：

如果成功，则返回一个新的 SOCKET 数据类型的套接字描述符。

如果失败，则返回一个 INVALID_SOCKET，错误信息可以通过 WSAGetLastError 函数返回。

参数说明如下。

af：指定地址簇，对于 TCP IP 协议的套接字，它只能写成 AF_INET(或 PF_INET)。

type：指定套接字类型，它只支持两种套接字：SOCK_STREAM(流式套接字)，SOCK_DGRAM(数据报式套接字)。

protocol：指定与特定地址家族相关的协议，如果指定为 0，那么系统就会根据地址格式和套接字类别，自动选择一个合适的协议。

3. 绑定端口(服务器)

函数原型：

```
int bind (
SOCKET s,
const struct sockaddr FAR * name,
int namelen
);
```

返回值：

如果成功，返回 0；如果失败，返回 SOCKET_ERROR，错误信息可以通过 WSAGetLastError 函数返回。

参数说明如下。

s：指定要绑定的套接字。

name：指定该套接字的本地地址信息，指向 SOCKADDR 结构体的指针。

namelen：指定该地址结构的长度。

4. 点分十进制转换成无符号长整型

函数原型：

```
unsigned long inet_addr (const char FAR * cp );
```

返回值：

一个对应 cp 点分十进制的 unsigned long 类型的数值。

参数说明如下。

cp：一个点分十进制的 IP 地址形式的字符串。

5. 无符号长整型转换成点分十进制

函数原型：

```
char FAR * inet_ntoa (struct in_addr in );
```

返回值：

一个对应 in 点分十进制的 IP 地址形式的字符串。

参数说明如下。

in：一个点分十进制的 unsigned long 类型的数值。

6. 主机字节顺序转换为网络字节顺序 16 位数值

函数原型：

```
u_short htons (u_short hostshort );
```

返回值：

把一个 u_short 类型的值从主机字节顺序转换为网络字节顺序 32 位数值。

参数说明：

hostshort：一个以主机字节顺序表示的 16 位数值。

函数原型：

```
u_long htonl (u_long hostlong );
```

返回值：

把一个 u_long 类型的值从主机字节顺序转换为网络字节顺序。

参数说明如下。

hostlong：一个以主机字节顺序表示的 32 位数值。

7. 相关结构体及宏

```
struct WSAData {
WORD wVersion,
 WORD wHighVersion,
 char szDescription[WSADESCRIPTION_LEN + 1],
char szSystemStatus[WSASYSSTATUS_LEN + 1],
unsigned short iMaxSockets,
unsigned short iMaxUdpDg,
char FAR * lpVendorInfo
};
```

参数说明如下。

wVersion：打算使用的 WinSock 版本号。

wHighVersion：容纳的是现有的 WinSock 最高版本号，以高字节代表的是 WinSock 的副版本号，低字节表示的是高版本号。

//以下两个参数一般不设置

```
unsigned short iMaxSockets：同时最多可以打开多少套接字
unsigned short iMaxUdpDg：数据报的最大长度
//同时最多可以打开套接字数目很大程度上和可用物理内存的多少有关
```

lpVendorInfo：这个参数是 WinSock 实施方案有关的指定厂商信息预留的，任何一个 Win32 平台上都没有使用这个字段。

8．主要数据结构

1）地址结构

```
struct sockaddr {
unsigned short sa_family,
 char sa_data[14]
 };
```

参数说明如下。

sa_family：指定地址家族，对于 TCP/IP 协议的套接字，必须设置为 AF_INET。

sa_data：仅表示要求一块内存分配区，起到占位的作用，该区域中指定与协议相关的具体地址信息，由于实际要求的只是内存区，所以对于不同的协议家族，用不同的协议家族，用不同的结构来替换 sockaddr。

2）TCP/IP 地址结构

```
struct sockaddr_in{
short sin_family,
unsigned short sin_port,
struct in_addr sin_addr,
char sin_zero[8]
};  //在 TCP/IP 编程中用这个结构体替换 sockaddr 结构体地址表示
```

参数说明如下。

sin_family：指定地址家族，对于 TCP/IP 协议的套接字，必须设置为 AF_INET。

sin_port：指定要分配给套接字的端口。

sin_addr：套接字的主机的 IP 地址。

sin_zero：一个填充占位符。

```
struct in_addr {
union {
struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b,
struct { u_short s_w1,s_w2; } S_un_w,
u_long S_addr,
} S_un;
};  //用于记入地址的一个结构，若将 IP 地址设置为 INADDR_ANY，允许套接字向任何分配给本机的
    //IP 地址发送或接收数据，用 INADDR_ANY 可以简化编程，这样程序便可以接收发自多个接口的
    //回应
```

3）MAKEWORD 宏

```
MAKEWORD(x,y);
```

功能：用于设置 DWORD 类型的版本号，x 是高字节，y 是低字节。

2.2.2 基于消息套接字编程相关函数

1. 获得系统中安装的网络协议的相关信息

函数原型：

```
int WSAEnumProtocols(
LPINT lpiProtocols,
LPWSAPROTOCOL_INFO lpProtocolBuffer,
ILPDWORD lpdwBufferLength
);
```

返回值：

如果函数没有错误发生，函数返回协议报告信息；如果错误，返回 SOCKET_ERROR 和在 WSAGetLastError 函数中查询相关详细信息。

参数说明如下。

lpiProtocols：一个以 NULL 结尾的协议标识号数组。这个参数是可选的，如果 lpiProtocols 为 NULL，则返回所有可用协议的信息，否则，只返回数组中列出的协议信息。

lpProtocolBuffer[out]：一个用 WSAPROTOCOL_INFO 结构体填充的缓冲区。WSAPROTOCOL_INFO 结构体用来存放或得到一个指定协议的完整信息。

lpdwBufferLength[in, out]：在输入时，指定传递给 WSAEnumProtocols() 函数的 lpProtocolBuffer 缓冲区的长度；在输出时，获取的所有请求信息需传递给 WSAEnumProtocols() 函数的最小缓冲区长度。这个函数不能重复调用，传入的缓冲区必须足够大以便能存放所有的元素。这个规定降低了该函数的复杂度，并且由于一个机器上装载的协议数目往往是很少的，所以并不会产生问题。

注意：Win32 平台支持多种不同的网络协议，采用 WinSock2，就可以编写可直接使用任何一种协议的网络应用程序了。通过 WSAEnumProtocols 函数可以获得系统中安装的网络协议的相关信息。

2. 注册网络事件

函数原型：

```
int WSAAsyncSelect (
SOCKET s,
HWND hWnd,
unsigned int wMsg,
long lEvent
);
```

返回值：

如果函数成功则返回值是 0，如果函数失败则返回值是 SOCKET_ERROR 并调用 WSAGetLastError 获得更多错误信息。

参数说明如下。

s：标识请求网络事件通知的套接字。

hWnd：指定网络事件发生时接收消息的窗口句柄。

wMsg：指定网络事件发生时窗口接收的消息。

lEvent：指定应用程序感兴趣的网络事件，可以是下面的组合取值。

相应取值含义如下。

FD_READ：应用程序想接收有关是否可读的通知。

FD_WRITE：应用程序想接收有关是否可写的通知。

FD_OBB：应用程序想要接收是否外带(OBB)数据抵达的通知。

FD_ACCEPT：应用程序想要接收与进入连接有关的通知。

FD_CONNECT：应用程序想要接收连接操作已完成的通知。

FD_CLOSE：应用程序想要接收与套接字关闭有关的通知。

FD_QOS：应用程序想要接收套接字"服务质量"发生更改的通知。

FD_GROUP_QOS：应用程序想要接收套接字组"服务质量"发生更改的通知。

FD_ROUTING_INTERFACE_CHANGE：应用程序想要接收在指定的方向上，与路由接口发生变化有关的通知。

FD_ADDRESS_LIST_CHANGE：应用程序想要接收、针对套接字的协议家族、本地地址列表发生变化的通知。

注意：该函数为指定的套接字请求基于 Windows 消息的网络事件通知，并自动将该套接字设置为非阻塞模式。

3. 创建套接字

函数原型：

```
SOCKET WSASocket(
int af,
int type,
int protocol,
LPWSAPROTOCOL_INFO lpProtocolInfo,
GROUP g,
DWORD dwFlags
);
```

返回值：

如果成功，则返回一个新的 SOCKET 数据类型的套接字描述符；

如果失败，则返回一个 INVALID_SOCKET，错误信息可以通过 WSAGetLastError 函数返回。

参数说明如下。

af：指定地址簇，对于 TCP/IP 协议的套接字，它只能写成 AF_INET(或 PF_INET)。

type：指定套接字类型，它只支持两种套接字，SOCK_STREAM(流式套接字)，SOCK_DGRAM(数据报式套接字)。

protocol：指定与特定地址家族相关的协议，如果指定为 0，那么系统就会根据地址格式和套接字类别，自动选择一个合适的协议。

lpProtocolInfo：一个指向 WSAPROTOCOL_INFO 结构体的指针，该结构定义了所创

建的套接字的特性。如果 lpProtocolInfo 为 NULL，则 WinSock2 DLL 使用前三个参数来决定使用哪一个服务提供者，它选择能够支持规定的地址族、套接字类型和协议值的第一个传输提供者。如果 lpProtocolInfo 不为 NULL，则套接字绑定到与指定的结构 WSAPROTOCOL_INFO 相关的提供者。

g：保留。

dwFlags：套接字属性的描述。

2.2.3 MFC 常用函数

初始化套接字 AfxSocketInit()函数原型如下：

```
BOOL AfxSocketInit(WSADATA * lpwsaData = NULL);
```

功能：MFC 提供的创建套接字库的函数。

返回值：若函数调用成功时，返回非零值，否则返回零。

注意：应该在应用程序类重载的 InitInstance()函数中调用 AfxSocketInit()函数，在 MFC 应用程序运行时需要在一些必要的预编译头文件 stdafx.h 中添加函数的头文件 afxsock.h。

优点：使用这个函数的优点是它可以确保在应用程序终止前，调用 WSACleapup 函数以终止对套接字的使用，并且利用 AfxSocketInit 函数也不用在加载套接字库时，手动为工程添加到 ws2_32.lib 的链接库文件设置。

2.2.4 TCP 套接字相关函数

1. 监听请求(服务器)

函数原型：

```
int listen (
SOCKET s,
int backlog
);
```

参数说明如下。

s：指用于监听用的套接字。

backlog：等待队列的最大长度，如果设置为 SOMAXCONN，那么下层的服务提供者将负责将这个套接字设置为最大的合理值(设置这个值是为了设置等待连接队列的最大长度，而不是在一个端口上同时可以进行连接的数目，例如，将 backlog 设置为 2，当有三个请求同时到达时，前两个连接请求会被放到请求连接队列中，然后由应用程序依次为这些请求服务，而第三个请求就被拒绝了)。

2. 接收请求(服务器)

函数原型：

```
SOCKET accept (
SOCKET s,
```

```
struct sockaddr FAR * addr,
int FAR * addrlen
);
```

返回值：如果没有错误发生，函数返回一个建立好连接的 SOCKET 套接字；如果失败则返回 INVALID_SOCKET。

参数说明如下。

s：指用于监听的套接字。

addr：指向一个缓冲区的指针，该缓冲区用来接收连接实体的地址，也就是当客户端向服务器发起的连接，服务器接收这个连接时，保存发起连接的这个客户端的 IP 地址信息和端口信息。

addrlen：一个返回值，指向一个整型，返回包含地址信息的长度。

3. 发送数据

函数原型：

```
int send (
SOCKET s,
const char FAR * buf,
int len,
int flags
);
```

参数说明如下。

s：指向一个已建立连接的套接字。

buf：指向一个缓冲区的指针，该缓冲包含将要传递的数据。

len：缓冲区的长度。

flags：设定的值将影响函数行为，一般设置为 0，MSG_PEEK 会使有用的数据被复制到接收缓冲区内，但没有从系统缓冲区中将其删除，MSG_OOB 表示处理带外数据。

4. 接收数据

函数原型：

```
int recv (
SOCKET s,
char FAR * buf,
int len,
int flags
);
```

参数说明如下。

s：指向一个已建立连接的套接字。

buf：指向一个缓冲区的指针，该缓冲用来接收保存数据。

len：缓冲区的长度。

flags：设定的值将影响函数行为，一般设置为 0，MSG_PEEK 会使有用的数据被复制到接收缓冲区内，但没有从系统缓冲区中将其删除，MSG_OOB 表示处理带外数据。

5. 建立连接(客服端)

函数原型:

```
int connect (
SOCKET s,
const struct sockaddr FAR * name,
int namelen
);
```

参数说明如下。

s:用来建立连接的套接字。

name:设定连接的服务器端的地址信息。

namelen:指定服务器端地址信息的长度。

2.2.5 UDP套接字相关函数

1. 接收数据

函数原型:

```
int recvfrom (
SOCKET s,
char FAR * buf,
int len,
int flags,
struct sockaddr FAR * from,
int FAR * fromlen
);
```

返回值:

如果没有错误,返回值是接收数据的字节数;如果连接关闭则返回0,否则返回SOCKET_ERROR。

参数说明如下。

s:准备接收数据的套接字。

buf:指向一个缓冲区的指针,该缓冲包含将要传递的数据。

len:缓冲区的长度。

flags:设定的值将影响函数行为,一般设置为0。

from:一个指向地址结构类型的指针,主要是用来接收发送数据方的地址信息。

fromlen:它是一个in out类型参数,表明在调用时需要给它一个初始值,当函数调用成功后,会通知这个参数返回一个值,返回值是地址结构的大小。

2. 发送数据

函数原型:

```
int sendto (
SOCKET s,
const char FAR * buf,
```

```
int len,
int flags,
const struct sockaddr FAR * to,
int tolen
);
```

返回值：

如果没有错误，返回值是发送数据的字节数，否则返回 SOCKET_ERROR。

参数说明如下。

s：准备发送数据的套接字。

buf：指向一个缓冲区的指针，该缓冲区用来接收保存数据。

len：缓冲区的长度。

flags：设定的值将影响函数行为，一般设置为 0。

to：一个指向地址结构类型的指针，主要是用来要指定目标套截止的地址。

tolen：指定目标套接字的地址的长度。

3. 消息接收(基于消息机制)

函数原型：

```
int WSARecvFrom(
SOCKET s,
LPWSABUF lpBuffers,
DWORD dwBufferCount,
LPDWORD lpNumberOfBytesRecvd,
LPDWORD lpFlags,
struct sockaddr FAR * lpFrom,
LPINT lpFromlen,
LPWSAOVERLAPPED lpOverlapped,
LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

参数说明如下。

s：标识套接字的描述符。

lpBuffers[in, out]：一个指向 WSABUF 结构体的指针。每一个 WSABUF 结构体包含一个缓冲区的指针和缓冲区的长度，WSABUF 结构体定义如下。

```
typedef struct __WSABUF {
u_longlen;                          //缓冲区长度
char FAR * buf;                     //指向缓冲区的指针
} WSABUF, FAR * LPWSABUF;
```

dwBufferCount：lpBuffers 数组中 WSABUF 结构体的数目。

lpNumberOfBytesRecvd[out]：如果接收操作立即完成，则为一个指向本次调用所接收的字节数的指针。

lpFlags[in, out]：一个指向标志位的指针，可以是以下组合值。相应取值含义如下。

MSG_PEEK：浏览到来的数据，这些数据将复制到缓冲区，但并不从输入队列中移除，

此标记仅对非重叠套接字有效。

MSG_OBB：处理外带(OBB)数据。

MSG_PARTIAL：此标记仅用于面向消息的套接字，作为输出参数时，此标记表明数据是发送方传送的一部分，消息的剩余部分将在随后的接收操作中被传送，如果随后的某个接收操作没有此标志，就表明这时发送方发送消息的尾部，作为输入参数时，此标记表明接收操作是完成的，即使只是一条消息部分数据已被服务提供者所接收。

lpFrom[out]：可选指针，指向重叠操作完成后存放源地址的缓冲区。

lpFromlen[in, out]：指向 from 缓冲区大小的指针，仅当指定了 lpFrom 才需要。

lpOverlapped：一个指向 WSAOVERLAPPED 结构体的指针(对于非重叠套接字则忽略)。

lpCompletionRoutine：一个指向接收操作完成时调用的完成例程的指针(对于非重叠套接字则忽略)。

如果创建的是重叠套接字，在使用函数时，一定要注意后面两个参数值，因为这时采用重叠 IO 操作，函数会立即返回，但接收到数据这一操作完成以后操作系统会调用 lpCompletionRoutine 参数指定的例程类通知调用进程。

函数原型：

```
void CALLBACK CompletionROUTINE(
DWORD dwError,
DWORD cbTransferred,
 LPWSAOVERLAPPED lpOverlapped,
 DWORD dwFlags
);
```

返回值：

若函数失败则返回 SOCKET_ERROR。

参数说明如下。

dwError：标志投递的重叠操作，比如 WSARecv，完成的状态是什么。

cbTransferred：指明了在重叠操作期间，实际传输的字节量是多大。

lpOverlapped：指明传递到最初的 IO 调用内的一个重叠结构。

dwFlags：返回操作结束时可能用的标志(一般没用)。

4. 消息发送(基于消息机制)

函数原型：

```
int WSASendTo(
SOCKET s,
LPWSABUF lpBuffers,
DWORD dwBufferCount,
LPDWORD lpNumberOfBytesSent,
DWORD dwFlags,
const struct sockaddr FAR * lpTo,
int iToLen,
LPWSAOVERLAPPED lpOverlapped,
```

```
LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

返回值：

若函数失败则返回 SOCKET_ERROR。

参数说明如下。

s：标识一个套接字(可能已连接)的描述符。

lpBuffers：一个指向 WSABUF 结构体的指针。每一个 WSABUF 结构体包含一个缓冲区的指针和缓冲区的长度。

dwBufferCount：lpBuffers 数组中 WSABUF 结构体的数目。

lpNumberOfBytesSent[out]：如果发送操作立即完成，则为一个指向本次调用所发送的字节数的指针。

dwFlags：指示影响操作行为的标志位。

lpTo：可选指针，指向目标套接字的地址。

iToLen：lpTo 中地址的长度。

lpOverlapped：一个指向 WSAOVERLAPPED 结构的指针(对于非重叠套接字则忽略)。

lpCompletionRoutine：一个指向接收操作完成时调用的完成例程的指针(对于非重叠套接字则忽略)。

2.2.6　编写套接字通信

1. 编写基于 UDP 套接字的通信

基于 UDP 套接字的通信流程如图 2-7 所示。

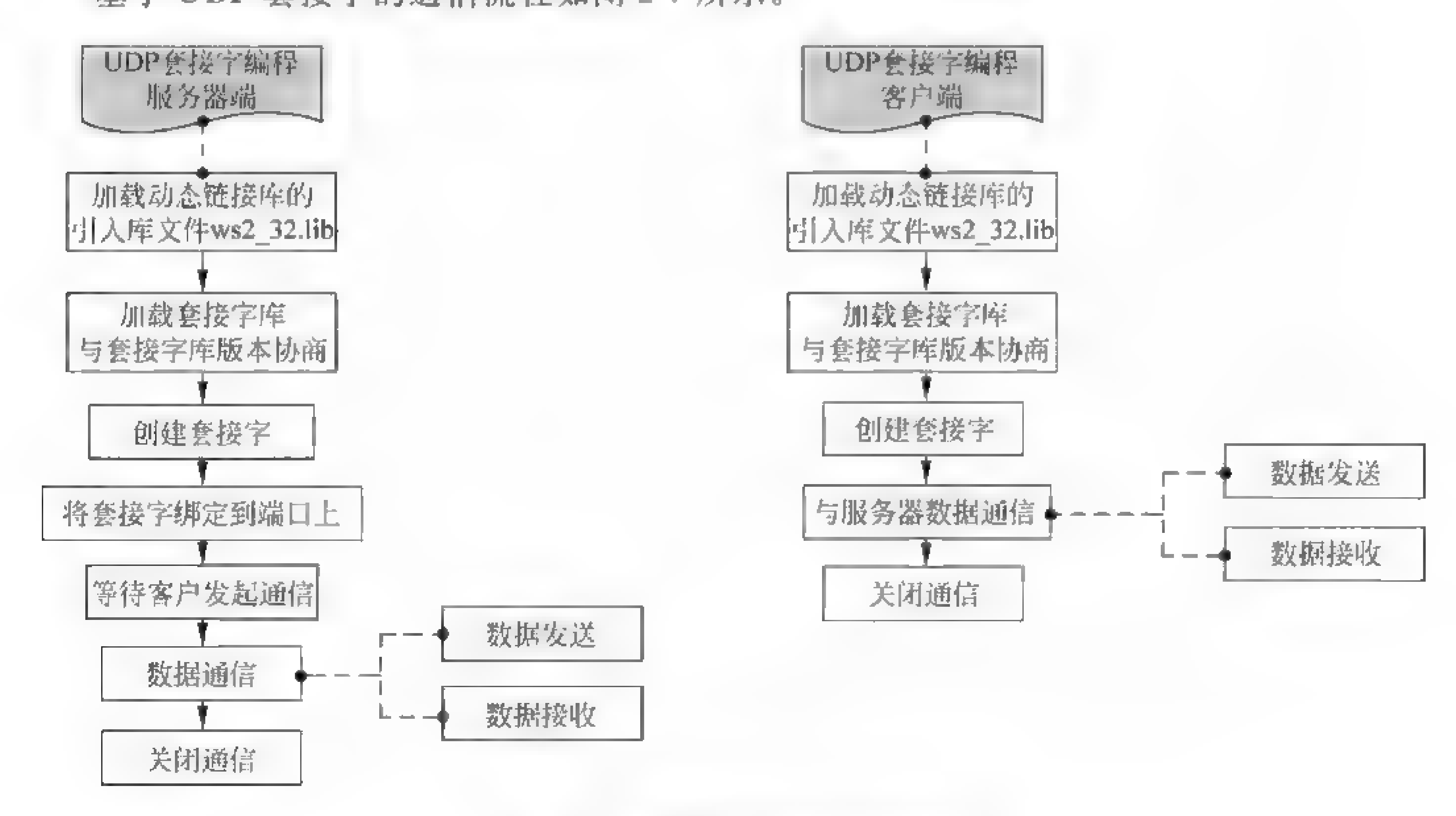


图 2-7　基于 UDP 套接字的通信

根据上述过程，实现代码如下。

(1) 查看本机 IP。

```
c:\>ipconfig

Windows IP 配置

无线局域网适配器 本地连接* 2:

   媒体状态  . . . . . . . . . . . . : 媒体已断开连接
   连接特定的 DNS 后缀 . . . . . . . :

无线局域网适配器 WLAN:

   连接特定的 DNS 后缀 . . . . . . . :
   本地链接 IPv6 地址. . . . . . . . : fe80::dd5b:68f5:268a:9f24%2
   IPv4 地址 . . . . . . . . . . . . : 10.171.76.186
   子网掩码  . . . . . . . . . . . . : 255.255.192.0
   默认网关. . . . . . . . . . . . . : 10.171.64.1

隧道适配器 Teredo Tunneling Pseudo-Interface:

   连接特定的 DNS 后缀 . . . . . . . :
   IPv6 地址 . . . . . . . . . . . . : 2001:0:9d38:6ab8:ce5:3097:7330:5dd9
   本地链接 IPv6 地址. . . . . . . . : fe80::ce5:3097:7330:5dd9%15
   默认网关. . . . . . . . . . . . . : ::
```

(2) 服务器端。

```
#include <Winsock2.h>
#include <iostream>
#include <string>
#pragma comment(lib,"ws2_32.lib")
using namespace std;

void main()
{
    //加载套接字库
    WORD wVersionRequested;
    WSADATA wsaData;
    int err;
    wVersionRequested = MAKEWORD( 1, 1 );
    err = WSAStartup( wVersionRequested, &wsaData );   //该函数的功能是加载一个 WinSocket
                                                       //库版本
    if ( err != 0 ) {
        return;
    }
    if ( LOBYTE( wsaData.wVersion ) != 1 ||
        HIBYTE( wsaData.wVersion ) != 1 ) {
            WSACleanup( );
            return;
    }
    //创建套接字
    SOCKET sockSrv = socket(AF_INET,SOCK_DGRAM,0);
    SOCKADDR_IN addrSrv;
```

```
    addrSrv.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
    addrSrv.sin_family = AF_INET;
    addrSrv.sin_port = htons(6000);
    //将套接字绑定到端口上
    bind(sockSrv,(SOCKADDR * )&addrSrv,sizeof(SOCKADDR));
    SOCKADDR_IN addrClient;
    int len = sizeof(SOCKADDR);
    char recvBuffer[300];                //接收字符数据
    memset((void * )recvBuffer,'\0',300);
    cout <<"等待对方发送数据..."<< endl;
    //接收数据
    recvfrom(sockSrv,recvBuffer,300,0,(SOCKADDR * )&addrClient,&len);
    cout <<"对方的地址为:"<< inet_ntoa(addrClient.sin_addr)<< endl;
    cout <<"接收的内容为:"<< recvBuffer << endl;
    //发送数据
    string sendBuffer = "this is server";
    cout <<"向客户端方发送数据:"<< sendBuffer.c_str()<< endl;
    sendto(sockSrv,sendBuffer.c_str(),sendBuffer.length() + 1,0,(SOCKADDR * )&addrClient,
sizeof(SOCKADDR));
    closesocket(sockSrv);                //关闭服务器套接字
    WSACleanup();                        //结束套接字库的调用
    system("pause");
}
```

(3) 客户端。

```
#include < Winsock2.h >
#include < iostream >
#include < string >
//加载动态连接库 ws2_32.dll,提供了网络相关 API 的支持
#pragma comment(lib,"ws2_32.lib")
using namespace std;
void main()
{
    //加载套接字库
    WORD wVersionRequested;
    WSADATA wsaData;
    int err;
    wVersionRequested = MAKEWORD( 1, 1 );
    err = WSAStartup( wVersionRequested, &wsaData ); //该函数的功能是加载一个 WinSocket
                                                     //库版本
    if ( err != 0 ) {
        return;
    }
    if ( LOBYTE( wsaData.wVersion ) != 1 ||
```

```
            HIBYTE( wsaData.wVersion ) != 1 ) {
                WSACleanup( );
                return;
        }
        //建立通信 Socket
        SOCKET sockClient = socket(AF_INET,SOCK_DGRAM,0);
        SOCKADDR_IN addrSrv;
        addrSrv.sin_addr.S_un.S_addr = inet_addr("10.171.76.185");
        addrSrv.sin_family = AF_INET;
        addrSrv.sin_port = htons(6000);
        //发送数据
        string sendBuffer = "this is client!";
        cout <<"向服务器方发送数据: "<< sendBuffer.c_str()<< endl;

sendto(sockClient,sendBuffer.c_str(),sendBuffer.length() + 1,0,(SOCKADDR * )&addrSrv,sizeof
(SOCKADDR));
        //接收数据
        char recvBuffer[300];                        //接收字符数据
        memset((void * )recvBuffer,'\0',300);
        int len = sizeof(SOCKADDR);
        cout <<"等待对方发送数据... "<< endl;
        recvfrom(sockClient,recvBuffer,300,0,(SOCKADDR * )&addrSrv,&len);
        cout <<"主机的地址为: "<< inet_ntoa(addrSrv.sin_addr)<< endl;
        cout <<"接收的内容为: "<< recvBuffer << endl;
        //结束通信
        closesocket(sockClient);                     //关闭服务器套接字
        WSACleanup();                                //结束套接字库的调用
system("pause");
}
```

(4) 运行结果如图 2-8 所示(上面为服务器端,下面为客户端)。



图 2-8 基于 UDP 的套接字通信运行结果图

2. 编写基于 TCP 的套接字通信

基于 TCP 的套接字通信流程如图 2-9 所示。

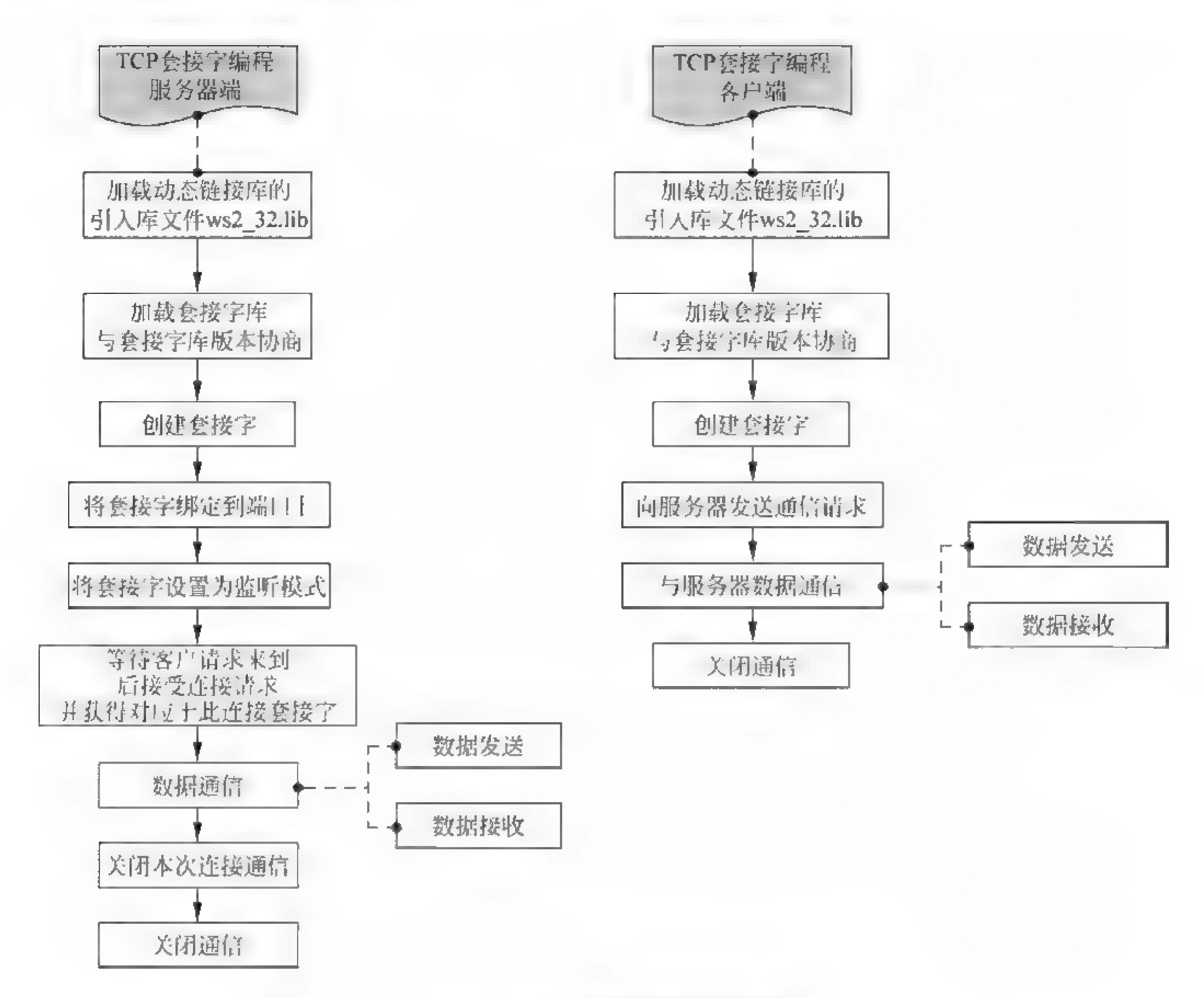


图 2-9 基于 TCP 的套接字通信

根据上述过程，基于 TCP 的套接字通信实现代码如下。

(1) 查看本机 IP。



(2) 服务器端。

```
#include "stdafx.h"
#include <Winsock2.h>
#include <Ws2tcpip.h>
#include <iostream>
#include <string>
#pragma comment(lib,"ws2_32.lib")
using namespace std;
void main()
{
    //加载套接字库
    WORD wVersionRequested;
    WSADATA wsaData;
    int err;
    wVersionRequested = MAKEWORD( 1, 1 );
    err = WSAStartup( wVersionRequested, &wsaData );  //该函数的功能是加载一个 WinSocket
                                                        //库版本
    if ( err != 0 ) {
        return;
    }
    if ( LOBYTE( wsaData.wVersion ) != 1 ||
        HIBYTE( wsaData.wVersion ) != 1 ) {
          WSACleanup( );
          return;
    }
    //创建套接字
    SOCKET sockSrv = socket(AF_INET,SOCK_STREAM,0);
    SOCKADDR_IN addrSrv;
    addrSrv.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
    addrSrv.sin_family = AF_INET;
    addrSrv.sin_port = htons(6000);
    //将套接字绑定到端口上
    bind(sockSrv,(SOCKADDR*)&addrSrv,sizeof(SOCKADDR));
    //将套接字设置为监听模式
    listen(sockSrv,5);
    //等待客户请求来到,当请求来到时候,接受请求,接受连接请求,返回一个新的对应于此连
    //接的套接字
    SOCKADDR_IN addrClient;
    int len = sizeof(SOCKADDR);
    //开始监听
    cout<<"等待用户连接"<<endl;
    SOCKET sockConn = accept(sockSrv,(SOCKADDR*)&addrClient,&len);  //sockConn 用于建立
                                                                      //连接的套接字
    cout<<"用户连接到来"<<endl;
    //和客户通信
    //接收数据
char recvBuffer[300];                                                 //接收字符数据
char sendBuf[20] = { '\0'};
    memset((void*)recvBuffer,'\0',300);
```

```
    cout <<"等待对方发送数据..."<< endl;
    recv(sockConn,recvBuffer,100,0);
    cout <<"对方的地址为: "<< inet_ntop(AF_INET, (void * )&addrClient.sin_addr, sendBuf, 16)
<< endl;
    cout <<"接收的内容为: "<< recvBuffer << endl;
    //发送数据
    string sendBuffer = "this is server";
    cout <<"向客户端方发送数据: "<< sendBuffer.c_str()<< endl;
    send(sockConn,sendBuffer.c_str(),sendBuffer.size(),0);
    //关闭本次连接的通道
    closesocket(sockConn);
    closesocket(sockSrv);          //关闭服务器套接字
    WSACleanup();                  //结束套接字库的调用
system("pause");
```

(3) 客户端。

```
#include <Winsock2.h>
#include <iostream>
#include <string>
//加载动态连接库 ws2_32.dll,提供了网络相关 API 的支持
#pragma comment(lib,"ws2_32.lib")
using namespace std;
void main( )
{
    //加载套接字库
    WORD wVersionRequested;
    WSADATA wsaData;
    int err;
    wVersionRequested = MAKEWORD( 1, 1 );
    err = WSAStartup( wVersionRequested, &wsaData );  //该函数的功能是加载一个 WinSocket
                                                      //库版本
    if ( err != 0 ) {
        return;
    }
    if ( LOBYTE( wsaData.wVersion ) != 1 ||
        HIBYTE( wsaData.wVersion ) != 1 ) {
          WSACleanup( );
          return;
    }
    //建立通信 Socket
    SOCKET sockClient = socket(AF_INET,SOCK_STREAM,0);
    SOCKADDR_IN addrSrv;
    addrSrv.sin_addr.S_un.S_addr = inet_addr("10.171.76.185");
    addrSrv.sin_family = AF_INET;
    addrSrv.sin_port = htons(6000);
```

```
    //发出连接请求
    cout <<"请求与服务器连接"<< endl;
    if(connect(sockClient,(SOCKADDR * )&addrSrv,sizeof(SOCKADDR)) != SOCKET_ERROR)
    {
        cout <<"与服务器建立连接"<< endl;
        //和服务器通信
        //发送数据
        string sendBuffer = "this is client!";
        cout <<"向服务器方发送数据: "<< sendBuffer.c_str()<< endl;
        send(sockClient,sendBuffer.c_str(),sendBuffer.size(),0);
        //接收数据
        char recvBuffer[300];          //接收字符数据
        memset((void * )recvBuffer,'\0',300);
        int len = sizeof(SOCKADDR);
        cout <<"等待对方发送数据..."<< endl;
        recv(sockClient,recvBuffer,100,0);
        cout <<"主机的地址为: "<< inet_ntoa(addrSrv.sin_addr)<< endl;
        cout <<"接收的内容为: "<< recvBuffer << endl;
    }
    //结束通信
    closesocket(sockClient);           //关闭服务器套接字
    WSACleanup();                      //结束套接字库的调用
    system("pause");
}
```

(4) 运行结果如图 2-10 所示(上面为客户端,下面为服务器端)。



图 2-10 基于 TCP 的套接字通信运行结果图

3. 编写基于消息机制的 UDP 套接字通信

基于消息机制的 UDP 通信流程如图 2-11 所示。

根据上述过程,基于消息机制的 UDP 套接字通信实现如下。

(1) 先添加一个对话框工程,工程名为 Chat,因为此处服务器和客户端写在一起,所以只有一个工程。

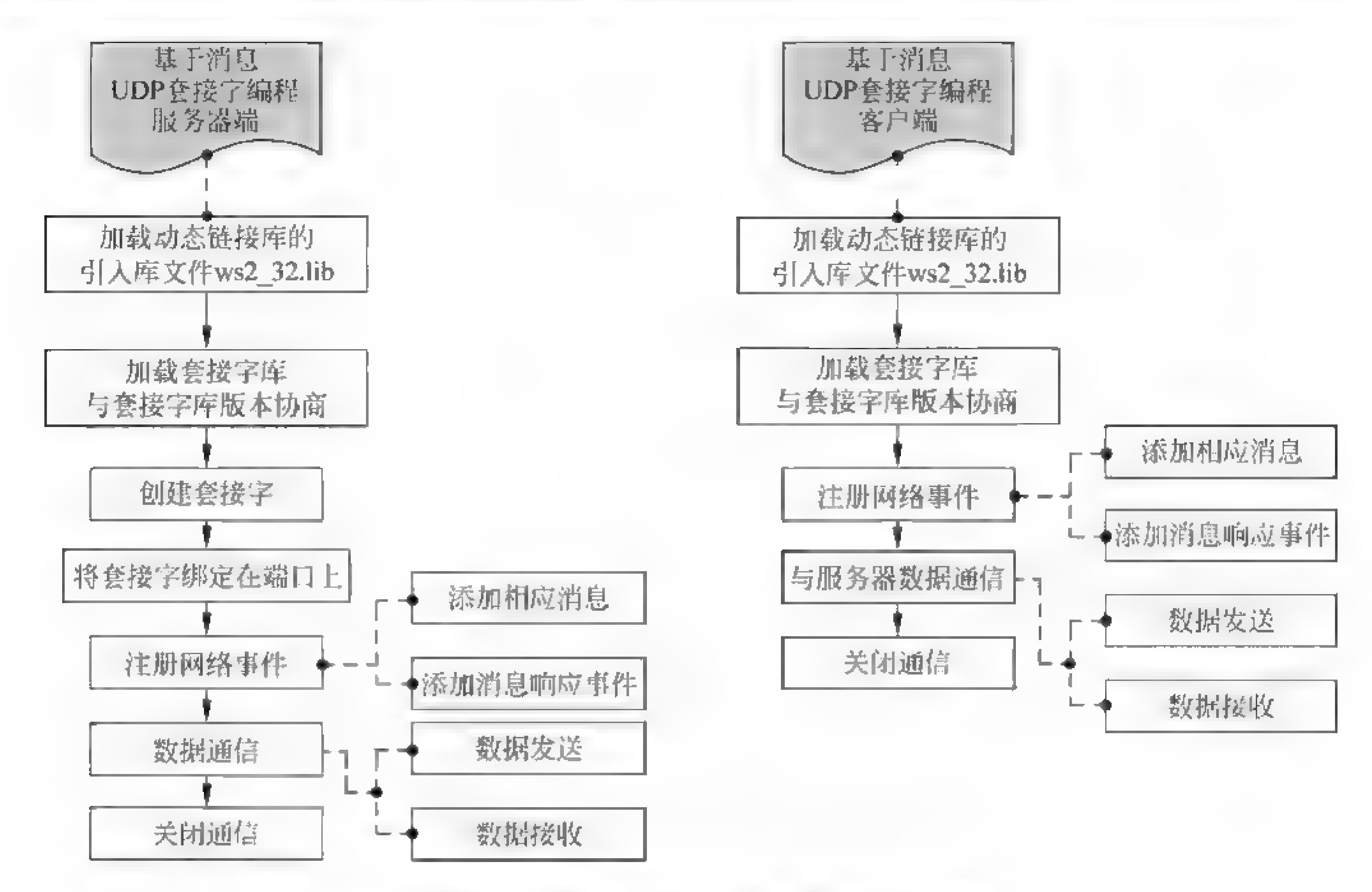


图 2-11　基于消息机制的 UDP 套接字通信

(2) 实现代码。

在 stdafx.h 头文件中加载动态链接库的引入库和相关头文件：

```
#include <winsock2.h>
#pragma comment(lib,"ws2_32.lib")
```

在程序初始化的时候，加载套接字库，并进行套接字库协商，这个工作放在 Chat.cpp 主线程的初始化工作函数 InitInstance 中。

```
BOOL CChatApp::InitInstance()
{
    //套接字版本协商-----------------------------------------
    WORD wVersionRequested;
    WSADATA wsaData;
    int err;
        wVersionRequested = MAKEWORD( 2, 2 );
        err = WSAStartup( wVersionRequested, &wsaData );
    if ( err != 0 ) {
            return FALSE;
    }
        if ( LOBYTE( wsaData.wVersion ) != 2 ||
        HIBYTE( wsaData.wVersion ) != 2 ) {
        WSACleanup( );
        return FALSE;
    }
    //------------------------------------------------------
    AfxEnableControlContainer();
```

```
    //Standard initialization
    //If you are not using these features and wish to reduce the size
    //of your final executable, you should remove from the following
    //the specific initialization routines you do not need.
#ifdef _AFXDLL
    Enable3dControls();         //Call this when using MFC in a shared DLL
#else
    Enable3dControlsStatic();  //Call this when linking to MFC statically
#endif
    CChatDlg dlg;
    m_pMainWnd = &dlg;
    int nResponse = dlg.DoModal();
    if (nResponse == IDOK)
    {
        //TODO: Place code here to handle when the dialog is
        //dismissed with OK
    }
    else if (nResponse == IDCANCEL)
    {
        //TODO: Place code here to handle when the dialog is
        //dismissed with Cancel
    }
    //Since the dialog has been closed, return FALSE so that we exit the
    //application, rather than start the application's message pump.
    return FALSE;
}
```

接着定义一个网络事件和网络事件响应函数：

```
/* 在 ChatDlg.h 中 */

//添加网络事件定义
#define UM_SOCK       WM_USER + 1
//然后在 CChatDlg 类中添加消息响应函数原型
afx_msg LRESULT OnSock(WPARAM,LPARAM);


/* 在 ChatDlg.cppp 中 */

//在消息映像中添加消息映像
BEGIN_MESSAGE_MAP(CChatDlg, CDialog)
    //{{AFX_MSG_MAP(CChatDlg)
    ON_WM_SYSCOMMAND()
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    ON_BN_CLICKED(IDC_BTN_SEND, OnBtnSend)
    //}}AFX_MSG_MAP
    ON_MESSAGE(UM_SOCK,OnSock)       //网络事件的消息映像
END_MESSAGE_MAP()

//添加消息事件响应函数定义
```

```
afx_msg LRESULT CChatDlg::OnSock(WPARAM wParam,LPARAM lParam)
{
    switch(LOWORD(lParam))
    {
    //接收数据
    case FD_READ:
        WSABUF wsabuf;
        wsabuf.buf = new char[200];
        wsabuf.len = 200;
        DWORD dwRead;
        DWORD dwFlag = 0;
        SOCKADDR_IN addrFrom;
        int len = sizeof(SOCKADDR);
        CString str;
        CString strTemp;
        HOSTENT * pHost;
        if(SOCKET_ERROR == WSARecvFrom(m_socket,&wsabuf,1,&dwRead,&dwFlag,
                      (SOCKADDR * )&addrFrom,&len,NULL,NULL))
        {
            MessageBox("接收数据失败!");
            return 0;
        }
        pHost = gethostbyaddr((char * )&addrFrom.sin_addr.S_un.S_addr,4,AF_INET);
        //str.Format(" %s 说 : %s",inet_ntoa(addrFrom.sin_addr),wsabuf.buf);
        str.Format(" %s 说 : %s",pHost -> h_name,wsabuf.buf);
        str += "\r\n";
        GetDlgItemText(IDC_EDIT_RECV,strTemp);
        str += strTemp;
        SetDlgItemText(IDC_EDIT_RECV,str);
        break;
    }
    return 0;
}
```

定义好网络事件及网络事件响应函数后，接着进行套接字的创建和端口绑定，以及网络事件注册，这个功能被封装在 InitSocket 中，并且在对话框初始化时就将调用它

```
/* 在 ChatDlg.h 中 */
//首先为 CChatDlg 类添加一个套接字成员变量，用于网络通信
SOCKET m_socket;
//然后在 CChatDlg 类中添加函数原型
BOOL InitSocket();
/* 在 ChatDlg.cppp 中 */
//定义 InitSocket 函数
BOOL CChatDlg::InitSocket()
{
    //套接字创建
    m_socket = WSASocket(AF_INET,SOCK_DGRAM,0,NULL,0,0);
    if(INVALID_SOCKET == m_socket)
    {
        MessageBox("创建套接字失败!");
```

```
        return FALSE;
    }

    //服务器端的端口绑定
    SOCKADDR_IN addrSock;
    addrSock.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
    addrSock.sin_family = AF_INET;
    addrSock.sin_port = htons(6000);
    if(SOCKET_ERROR == bind(m_socket,(SOCKADDR * )&addrSock,sizeof(SOCKADDR)))
    {
        MessageBox("绑定失败!");
        return FALSE;
    }
    //注册网络事件
    if(SOCKET_ERROR == WSAAsyncSelect(m_socket,m_hWnd,UM_SOCK,FD_READ))
    {
        MessageBox("注册网络读取事件失败!");
        return FALSE;
    }
    return TRUE;
}
//在定义"发送"按钮消息事件时让其能发送对应消息
void CChatDlg::OnBtnSend()
{
    //TODO: Add your control notification handler code here
    DWORD dwIP;
    CString strSend;
    WSABUF wsabuf;
    DWORD dwSend;
    int len;
    CString strHostName;
    SOCKADDR_IN addrTo;
    HOSTENT * pHost;
    //客户端
    //获取服务器端 IP 地址
    ((CIPAddressCtrl * )GetDlgItem(IDC_IPADDRESS1)) -> GetAddress(dwIP);
    addrTo.sin_addr.S_un.S_addr = htonl(dwIP);
    //绑定端口和设置 IP 协议
    addrTo.sin_family = AF_INET;
    addrTo.sin_port = htons(6000);
    //获得发送文本
    GetDlgItemText(IDC_EDIT_SEND,strSend);
    len = strSend.GetLength();
    wsabuf.buf = strSend.GetBuffer(len);
    wsabuf.len = len + 1;
    SetDlgItemText(IDC_EDIT_SEND,"");
    //发送数据
    if(SOCKET_ERROR == WSASendTo(m_socket,&wsabuf,1,&dwSend,0,
            (SOCKADDR * )&addrTo,sizeof(SOCKADDR),NULL,NULL))
```

```
    {
        MessageBox("发送数据失败!");
        return;
    }
}
```

(3) 运行结果如图 2-12 所示。

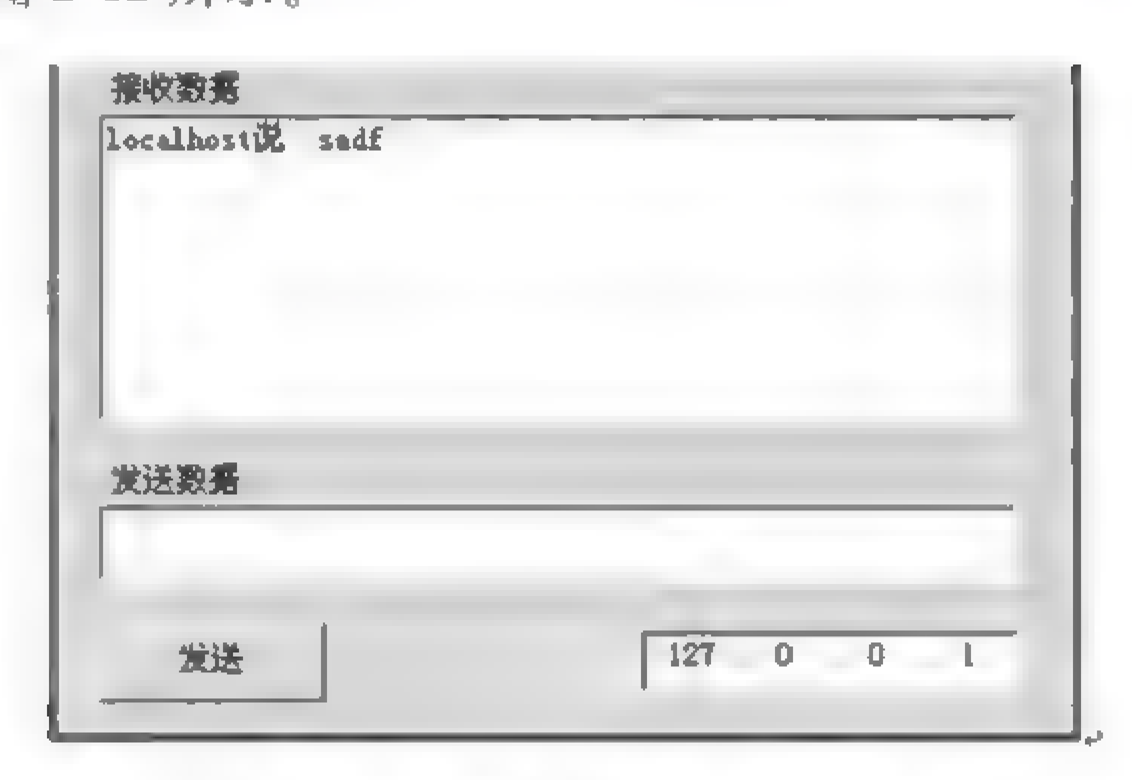


图 2-12　基于消息机制的 UDP 套接字通信运行结果图

4. 通过域名获得 IP 地址

在使用 Socket 程序之前必须调用 WSAStartup 函数，然后解析域名得到 IP 地址，实现代码如下。

(1) 代码示例。

```
#include <Winsock2.h>
#include <iostream>
#include <string>
#pragma comment(lib,"ws2_32.lib")
using namespace std;
void main()
{
    //加载套接字库
    WORD wVersionRequested;
    WSADATA wsaData;
    int err;
    wVersionRequested = MAKEWORD( 1, 1 );
    err = WSAStartup( wVersionRequested, &wsaData );   //该函数的功能是加载一个 WinSocket
                                                        //库版本
    if ( err != 0 ) {
         return;
    }
    if ( LOBYTE( wsaData.wVersion ) != 1 ||
         HIBYTE( wsaData.wVersion ) != 1 ) {
            WSACleanup( );
            return;
    }
```

```
    //解析域名获得 IP 地址
    hostent * pHostent = gethostbyname("www.baidu.com");
    sockaddr_in sa;
    ZeroMemory(&sa, sizeof(sa));
    //获得 IP 地址
    memcpy(&sa.sin_addr.s_addr,pHostent->h_addr_list[0],pHostent->h_length);
        //将 IP 地址转为字符串形式,输出 IP 地址
    string strTemp = inet_ntoa(sa.sin_addr);
    cout << strTemp << endl;

    //结束套接字库的调用
    WSACleanup();
    system("pause");
}
```

(2) 运行结果如图 2-13 所示。



图 2-13 根据域名获取 IP 地址运行结果图

在控制台中进行 ping www.baidu.com 的运行结果如图 2-14 所示。

```
c:\>ping www.baidu.com

正在 Ping www.a.shifen.com [119.75.218.70] 具有 32 字节的数据:
来自 119.75.218.70 的回复: 字节=32 时间=34ms TTL=52
来自 119.75.218.70 的回复: 字节=32 时间=43ms TTL=52
来自 119.75.218.70 的回复: 字节=32 时间=42ms TTL=52
来自 119.75.218.70 的回复: 字节=32 时间=45ms TTL=52

119.75.218.70 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间(以毫秒为单位):
    最短 = 34ms, 最长 = 45ms, 平均 = 41ms
```

图 2-14 ping www.baidu.com 运行结果图

2.3 Visual C++网络安全编程

C++语言在 C 语言的基础上,提供了面向对象的编程支持,是一种应用很广泛的编程语言,基于 Visual C++的网络安全编程可以提供信息安全防护,保证数据安全、系统安全、网络安全。接下来介绍 Visual C++网络安全编程所需的相关函数和方法。

2.3.1 获取系统实时信息

为了详细了解当前计算系统的实时信息,例如,计算机名、操作系统及其版本号、内存容量、系统目录、驱动器及分区类型等,可利用函数实现上述功能。具体实现函数以及实现代码如下。

```
#include "stdafx.h"
#include<stdio.h>
#include<windows.h>
#include<string.h>
#include<iostream>
void GetSysInfo();
DWORD GetOS()
{
OSVERSIONINFO os;
os.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
GetVersionEx(&os);
switch (os.dwPlatformId)
{
case VER_PLATFORM_WIN32_WINDOWS:
      return VER_PLATFORM_WIN32_WINDOWS;
case VER_PLATFORM_WIN32_NT:
      return VER_PLATFORM_WIN32_NT;
}
return 0;
}
VOID GetSysInfo()
{
TCHAR szBuff[MAX_PATH];
TCHAR szTemp[MAX_PATH];
wsprintf(szBuff, L"<<System Information>>\n\n\r");  //加 L 是为了将 TCHAR 型转换成 LCPwstr 型
wprintf(szBuff);                                     //因为 szBuff 是 tchar 字符型,所以用
                                                     //wprintf 输出,用 print 只能输出一个字符
                                                     //计算机名
DWORD len = sizeof(szTemp);
GetComputerName(szTemp, &len);
wsprintf(szBuff,L"Computer Name: %s\n\n\r", szTemp);
wprintf( szBuff);
//当前操作系统
switch (GetOS())
{
case VER_PLATFORM_WIN32_WINDOWS:
     lstrcpy(szTemp, L"Windows 9x");
     break;
case VER_PLATFORM_WIN32_NT:
     lstrcpy(szTemp, L"Windows NT/2000");
     break;
}
wsprintf(szBuff, L"Option System: %s\n\n\r", szTemp);
wprintf(szBuff);
//内存容量
MEMORYSTATUS mem;
mem.dwLength = sizeof(mem);
GlobalMemoryStatus(&mem);
wsprintf(szBuff, L"Total Memroy: %dM\n\n\r", mem.dwTotalPhys / 1024 / 1024 + 1);
wprintf(szBuff);
//系统目录
TCHAR szPath[MAX_PATH];
GetWindowsDirectory(szTemp, sizeof(szTemp));
```

```
GetSystemDirectory(szBuff, sizeof(szBuff));
wsprintf(szPath, L" Windows Directory:  % s/n/n/rSystem Directory:  % s\n\n\r", szTemp,
szBuff);
wprintf(szBuff);
//驱动器及分区类型
TCHAR szFileSys[10];
for (int i = 0; i<26; ++i)
{
     wsprintf(szTemp, L" % c://", 'A' + i);
     UINT uType = GetDriveType(szTemp);
     switch (uType)
     {
         case DRIVE_FIXED: GetVolumeInformation(szTemp, NULL, NULL, NULL, NULL, NULL, szFileSys,
MAX_PATH);
         wsprintf(szBuff, L"Hard Disk: % s ( % s)\n\n\r", szTemp, szFileSys);
         wprintf(szBuff);
         break;
    case DRIVE_CDROM:
         wsprintf(szBuff, L"CD - ROM Disk: % s\n\n\r", szTemp);
         wprintf(szBuff);
         break;
         case DRIVE_REMOTE: GetVolumeInformation(szTemp, NULL, NULL, NULL, NULL, NULL, szFileSys,
MAX_PATH);
         wsprintf(szBuff, L"NetWork Disk: % s ( % s)\n\n\r", szTemp, szFileSys);
         wprintf(szBuff);
         break;
    }
}
}
int main(void)
{
GetSysInfo(); return 0;
}
```

运行结果如图 2-15 所示。



图 2-15 获取系统实时信息运行结果图

2.3.2 进程处理

1. 进程定义

狭义定义：进程是正在运行的程序的实例。

广义定义：进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元。

进程的概念主要有两点：第一，进程是一个实体。每一个进程都有它自己的地址空间，一般情况下，包括文本区域、数据区域和堆栈。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储着活动过程调用的指令和本地变量。第二，进程是一个“执行中的程序”。程序是一个没有生命的实体，只有处理器赋予程序生命时（操作系统执行它），它才能成为一个活动的实体，我们称其为进程。

进程是操作系统中最基本、重要的概念，是多道程序系统出现后，为了刻画系统内部出现的动态情况，描述系统内部各道程序的活动规律引进的一个概念，所有多道程序设计操作系统都建立在进程的基础上。

2. 相关函数

创建进程可以使用函数 WinExec()、ShellExecute()以及 CreateProcess()。一般使用 WinExec()和 ShellExecute()创建设置比较简单的进程。

函数 WinExec()的定义如下。

```
UINT WinExec(LPCSTR lpCmdLine, UINT uCmdShow);
```

返回值：

若函数调用成功，则返回值大于 31。若函数调用失败，则有几种返回值，例如，0 表示系统内存或资源已耗尽，ERROR_BAD_FORMAT 表示 EXE 文件无效，ERROR_FILE_NOT_FOUND 表示指定的文件未找到，ERROR_PATH_NOT_FOUND 表示指定的路径未找到。

参数说明如下。

lpCmdLine：指向一个空结束的字符串，串中包含将要执行的应用程序的命令行。

uCmdShow：Windows 应用程序的窗口如何显示。

函数 ShellExecute()的定义如下。

```
HINSTANCE ShellExcute(HWND hwnd,
                LPCTSTR lpOperation,
                LPCTSTR lpFile,
                LPCSTR  lpParameters,
                LPCSTR  lpDirectory,
                 INT     nShowCmd);
```

返回值：

若函数调用成功，则返回值大于 32，否则为一个小于或等于 32 的错误值。

参数说明如下。

hwnd：指向父窗口的窗口句柄。

lpOperation：一个空结束的字符串地址，此字符串指定要执行的操作。一般情况下，字符串 Open 表示打开由参数 lpFile 指定的文件，字符串 Print 表示打印由参数 lpFile 指定的文件。

lpFile：一个空结束的字符串地址，此字符串指定要打开或打印的文件。

lpParameters：打开应用程序的参数。

lpDirectory：一个空结束的字符串地址，此字符串指定默认目录。参数 nShowCmd 表示应用程序打开时如何显示。

创建配置复杂的进程使用函数 CreateProcess 来实现，其定义如下。

```
BOOL CreateProcess(
        LPCTSTR lpApplicationName,
        LPTSTR lpCommandLine,
        LPSECURITY_ATTTRIBUTES lpProcessAttributes,
        LPSECURITY_ATTTRIBUTES lpThreadAttributes,
        BOOL bInheritHandles,
        DWORD dw CreationsFlags,
        LPVOID lpEnvironment,
        LPCTSTR lpCurrentDirectory,
        LPSTARTUPINFO lpStartupInfo,
        LPPROCESS_INFORMATION lpProcessInformation
);
```

参数说明如下。

IpApplicationName：可执行模块名。

lpCommandLine：命令行字符串。

lpProcessAtrributes：进程的安全属性。

lpThreadAttributes：线程的安全属性。

bInheritHnadles：句柄继承性质。

dwCreationFlags：创建标志。

lpEnvironment：指向新的环境块的指针。

lpCurrentDirectory：指向当前目录名的指针。

lpStartupInfo：指向启动信息结构的指针。

lpProcessInformation：指向进程信息结构的指针。

在这个函数中使用数据结构 PROCESS_INFORMATION，此数据结构描述了进程的一些信息，其结构体描述如下。

```
Typedef struct_PROCESS_INFORMATION
{
        HANDLE hProcess;
        HANDLE hThread;
        DWORD dwProcessId;
        DWORD dwThreadId;
}PROCESS_INFORMATION, * PROCESS_INFORMATION, * LPROCESS_INFORMATION;
```

返回值：

若函数调用成功，则返回值不为 0；若函数调用失败，则返回值为 0。

参数说明如下。

hProcess：存放每个对象的进程相关的句柄。

hThread：返回的线程句柄。

dwProcessId：存储进程 ID 号。

dwThreadId：存储线程 ID。

在执行 CreateProcess()函数后，这些信息会保存下来。

参数 lpStartupInfo 是 STRATUPINFO 结构，此结构体的内容可以用来设置新进程的标题，新窗口的初始大小和位置及复位向标准输入和输出。

2.3.3　线程处理

1. 线程定义

进程有很多优点，它提供了多道编程，让每个人感觉拥有自己的 CPU 和其他资源，可以提高计算机的利用率。但是，仔细观察就会发现进程还是有很多缺陷的，主要体现在两点上：首先，进程只能在一个时间干一件事；其次，进程在执行过程中如果阻塞，例如等待输入，整个进程就会挂起，即使进程中有些工作不依赖于输入的数据，也将无法执行。因此，实际操作系统中，引入了这种类似的机制——线程。

进程和线程的并发层次不同：进程属于在处理器这一层上提供的抽象；线程则属于在进程这个层次上再提供了一层并发的抽象。除了提高进程的并发度，线程还有一个好处，就是可以有效地利用多处理器和多核计算机。现在的处理器朝着多核方向发展，在没有线程之前，多核并不能让一个进程的执行速度提高，但如果将一个进程分解为若干个线程，则可以让不同的线程运行在不同的核上，从而提高了进程的执行速度。

具体而言，进程与线程的区别如下。

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，进程是系统进行资源分配和调度的一个独立单位。

线程是进程的一个实体，是 CPU 调度和分派的基本单位，它是比进程更小的能独立运行的基本单位。线程自己基本上不拥有系统资源，只拥有一点儿在运行中必不可少的资源（如程序计数器，一组寄存器和栈），但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。

一个线程可以创建和撤销另一个线程，同一个进程中的多个线程之间可以并发执行。

进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其他进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。

2. 相关函数

在 Windows 中利用线程 API 函数完成线程的创建、挂起、恢复、终结以及通信等任务。主要涉及的函数有：CreateThread()、SuspendThread()、ResumeThread()、ExitThread()、

TerminateThread()、PostThreadMessage()、SetThreadPriority()等。

1) 创建线程

创建线程使用函数 CreateThread(),其定义如下。

```
HANDLE CreateThread ( LPSECURITY_ATTRIBUTES lpThreadAttributes,
                      DWORD dwStackSize,
                      LPTHREAD_START_ROUTINE lpStartAddress,
                      LPVOID lpParameter,
                      DWORD dwCreationFlags,
                      LPDWORD lpThreadID);
```

返回值：

该函数会创建一个新的线程,并返回已建线程的句柄。如果函数执行成功就返回线程句柄,失败就返回 NULL。

参数说明如下。

lpThreadAttributes：指向一个 SECURITY_ATTRIBURES 结构的指针,该结构中定义线程的安全属性,一般置为 NULL。

dwStackSize：线程的堆栈深度,一般都设置为 0。参数 lpStartAddress 表示新线程开始执行时代码所在函数的地址,即线程的起始地址。一般情况为(LPTHREAD_START_ROUTINE)FuncName,FuncName 是线程函数名。

lpParameter：线程执行时传递给线程的 32 位参数,即线程函数的参数。

dwCreationFlags：控制线程创建的附加标志,如果该参数为 0,线程在被创建后就立即开始执行；如果该参数为 CREATE_SUSPENDED,则线程被创建后处于挂起状态,不立即执行,直至调用函数 ResumeThread()。

lpThread：所创建线程的 ID。

线程函数的类型如下所示。

```
DWORDWINAPI FuncName(LPVOIDlpv ThreadParm);
```

功能：

该函数输入一个 LPVOID 型的参数,可以是一个 DWORD 型的整数,也可以是一个指向一个缓冲区的指针,返回一个 DWORD 型的值。

2) 线程挂起与恢复

挂起线程使用函数 SuspendThread(),表示如下。

```
DWORD SuspendThread(HANDLE hThread);
```

功能：

该函数用于挂起指定的线程,如果函数执行成功,则线程的执行被暂停。

结束线程使用函数 ResumeThread(),表示如下。

```
DWORD ResumeThread(HANDLE hThread);
```

功能：

该函数用于恢复挂起的线程,继续执行线程。

3）终止线程

终止线程使用函数 ExitThread()，表示如下。

```
VOID ExitThread( DWORD dwExitCode)
```

功能：

该函数用于终止自身的执行，主要在线程的执行函数中被调用。

参数说明如下。

dwExitCode：设置线程的退出码。

另外一个终止线程的函数为 TerminateThread()，其定义如下。

```
BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode);
```

参数说明如下。

hThread：将被终结的线程的句柄。

dwExitCode：指定线程的退出码。

使用 TerminateThread()终止某个线程的执行是不安全的，可能会引起系统不稳定；虽然该函数会立即终止线程的执行，但并不释放线程所占用的资源。因此，一般不建议使用该函数。

一般情况下，线程运行结束之后，线程函数正常返回，但是应用程序可以调用 TermianteThread 强行终止某一线程的执行。

4）线程消息

线程之间传递消息可以使用函数 PostThreadMeaage()，其定义如下。

```
BOOL PostThreadMessage(DWORD idThread,
                       UINT Msg,
                       WPARAM wParam,
                       LPARAM IParam);
```

返回值：

该函数将一条消息放入到指定线程的消息队列中，并且不等到消息被该线程处理时便返回。调用该函数时，如果即将接收消息的线程没有创建消息循环，则该函数执行失败。

参数说明如下。

idThread：将接收消息的线程的 ID。

Msg：指定用来发送的消息。

wParam 和 IParam：同消息有关的参数。

5）线程优先级操作

设定线程的相对优先级使用函数 SetThreadPriority()，定义如下。

```
BOOL SetThreadPriority(HANDLE hThread, int nPriority);
```

返回值：

该函数成功执行则返回非 0，失败则返回 0，可以使用 GetLastError 获取信息。

参数说明如下。

hThread：指向待修改优先级线程的句柄。

nPriority：优先级，其可能的取值如下。

```
THREAD_PRIORITY_LOWEST,
THREAD_PRIORITY_BELOW_NORMAL,
THREAD_PRIORITY_NORMAL,
THREAD_PRIORITY_ABOVE_NORMAL,
THREAD_PRIORITY_HIGHEST
```

当一个线程被首次创建时，它的优先级等同于它所属进程的优先级。在单个进程内可以通过调用 SetThreadPriority()函数改变线程的相对优先级。一个线程的优先级是相对于其所属的进程的优先级而言的。

获取线程优先级可以使用函数 GetThreadPriority()，定义如下。

```
Int GetThreadPriority(
                    HANDLE hThread
);
```

返回值：

线程的优先级。

参数说明如下。

hThread：线程的句柄。

2.3.4 定时器处理

在 Windows 中，可以使用多种定时器操作，利用它们可以设置相对比较精确的定时器。在 Windows 中与定时器处理相关的函数有 CreateWaitableTimer()和 SetWaitableTimer()。

1. 相关函数

1) 创建定时器

创建定时器使用函数 CreateWaitableTimer()，定义如下。

```
HANDLE CreateWaitableTimer(
  LPSECURITY_ATTRIBUTES lpTimerAttributes,
  BOOL bManualReset,
  LPCTSTR lpTimerName
);
```

返回值：

如果函数执行成功就返回时间对象句柄；如果时间对象句柄已经存在，那么就返回已经存在的时间对象句柄，此时可以使用函数 GetLastError()返回代码之 ERROR_ALREADY_EXISTS，如果函数调用失败，返回值为 NULL，可以使用函数 GetLastError()返回具体信息。

参数说明如下。

lpTimerAttributes：定时器的属性。

bManualReset：是否手动复位。

lpTimerName：定时器的名称。

2）设置定时器

设置定时器使用函数 SetWaitableTimer()，定义如下。

```
BOOL SetWaitableTimer(
    HANDLE hTimer,
    Const LARGE_INTEGER * lpDueTime,
    LONG LPeriod,
    PTIMERAPCROUTINE pfnCompletionRoutine,
    LPVOID lpArgToCompletionRoutine,
    BOOL fResume
);
```

返回值：

该函数成功执行则返回非零，失败则返回 0，可以使用 GetLastError()获取信息。

参数说明如下。

hTimer：定时器句柄。

lpDueTime：时间间隔，如果是正值则是绝对时间，如果是负值则是相对时间。

LPeriod：周期。

pfnCompletionRoutine：回调函数。

lpArgToComletionRoutine：传递给回调函数的参数。

fResume：系统是否自动回复。

2. 应用举例

下面的例子说明如何创建定时器，以及如何设置定时器的时间。程序的功能是每隔 1s 就输出一个数，数字在不断递增。

```
#define _WIN32_WINNT 0X0500
#include "stdafx.h"
#include <stdio.h>
#include <conio.h>
#include <windows.h>
//int CreateTestTimer(void)
void main()
{
//定义时间对象句柄
HANDLE Timer = NULL;
LARGE_INTEGER TimerNumber;
//设置相对时间为 1s
TimerNumber.QuadPart = -10000000;
int KeyInfo;
while (1)
{
    //创建定时器
    Timer = CreateWaitableTimer(NULL, TRUE, TEXT("NewTimer"));
    if (Timer == NULL)
    {
```

```
            printf("CreateWaitableTimer with error % d/n", GetLastError());
            return ;
        }
        //设置
        if (!SetWaitableTimer(Timer, &TimerNumber, 0, NULL, NULL, 0))
        {
              printf("SetWaitableTimer with error % d/n", GetLastError());
              CloseHandle(Timer);
              return ;
        }
        //等待定时器
        if (WaitForSingleObject(Timer, INFINITE) != WAIT_OBJECT_0)
        {
            CloseHandle(Timer);
            return ;
        }
        else
        {
            static int number = 0;
            //定时器完成
            number++;
            printf(" % d\n",number);     //添了一句,可以在屏幕显示功能效果
        }//
    }
    }
```

运行结果如图 2-16 所示。



图 2-16 定时器运行结果图

2.3.5 注册表处理

1. 注册表概念

注册表(Registry)是Windows操作系统中的一个核心数据库,其中存放着各种参数,直接控制着Windows的启动、硬件驱动程序的装载以及一些Windows应用程序的运行,从而在整个系统中起着核心作用。这些参数包括软硬件的相关配置和状态信息,比如应用程序和资源管理器外壳的初始条件、首选项和卸载数据等,联网计算机的整个系统的设置和各种许可,文件扩展名与应用程序的关联,硬件部件的描述、状态和属性,性能记录和其他底层的系统状态信息,以及其他数据等。具体来说,在启动Windows时,Registry会对照已有硬件配置数据,检测新的硬件信息;系统内核从Resistry中选取信息,包括要装入什么设备驱动程序,装入次序,内核传送回它自身的信息,例如版权号等;同时设备驱动程序也向Registry传送数据,并从Registry接收装入和配置参数,例如硬件中断或DMA通道等;另外,设备驱动程序还要报告所发现的配置数据,为应用程序或硬件的运行提供增加新的配置数据的服务。同时Windows还提供了大量其他接口,允许用户修改系统配置数据,例如控制面板、设置程序等。

如果注册表受到了破坏,轻则使Windows的启动过程出现异常,重则可能会导致整个Windows系统的完全瘫痪。因此正确地认识、使用,特别是及时备份注册表对Windows用户来说就显得非常重要。

2. 注册表的数据结构与相关函数

注册表由键(也叫主键或称"项")、子键(子项)和值项构成。一个键就是分支中的一个文件夹,而子键就是这个文件夹当中的子文件夹,子键同样也是一个键。一个值项则是一个键的当前定义,由名称、数据类型以及分配的值组成。一个键可以有一个或多个值,每个值的名称各不相同,如果一个值的名称为空,则该值为该键的默认值。

在注册表编辑器(regedit.exe)中,数据结构显示如下,其中,command键是open键的子键,(默认)表示该值是默认值,值名称为空,其数据类型为REG_SZ,数据值为%systemroot%/system32/notepad.exe" %1数据类型。

注册表的数据类型主要有以下4种。

REG_SZ:文本字符串。

REG_MULTI_SZ:含有多个文本值的字符串。

REG_BINARY:二进制值,以十六进制显示。

REG_DWORD:一个32位的二进制值,显示为8位的十六进制值。

注册表操作设计的API函数主要由RegCreateKeyEx()、RegOpenKeyEx()、RegSetValueEx()、RegQueryValueEx()和RegCloseKey()几个函数实现。

1) 创建键

创建键由函数RegCreateKeyEx()完成,其定义如下。

```
LONG RegCreateKeyEx(HKEY hKey,
                    LPCSTR lpSubKey,
                    DWORD ulOptions,
```

```
                REGSAM  samDesired,
                PHKEY phkResult
);
```

返回值：

如果函数调用成功，返回 ERROR_SUCCESS；如果调用失败，返回一个非零错误码。

参数说明如下。

hKey：要打开的键的句柄，例如 HKEY_CLASS_ROOT、HKEY_CURRENT_CONFIG、HKEY_CURRENT_USER、HKEY_LOCAL_MACHINE 和 HKEY_USERS。

lpSubKey：要打开子键的名称。

ulOptions：保留，设为 0。

samDesired：打开方式。

phkResult：指向接收打开或新建键的变量。

2) 打开键

打开键使用函数 RegOpenKeyEx( )，其定义如下。

```
LONG RegOpenKeyEx(
               HKEY hKey,
               LPCTSTR lpSubKey,
               DWORD ulOptions,
               REGSAM samDesired,
               PHKEY phkResult
);
```

返回值：

如果函数调用成功，返回 ERROR_SUCCESS。如果函数调用失败，返回一个非零的错误代码。

参数说明如下。

hKey：已经打开的键的句柄。

lpSubKey：要打开的子键的名称。

ulOptions：保留，设为 0。

samDesired：打开方式。

phkResult：返回打开的子键的句柄。

3) 设置键值

设置键值使用函数 RegSetValueEx( )，其定义如下。

```
LONG RegSetValueEx(HKEY hKey,
                 LPCTSTR lpValueName,
                 LPDWORD lpReserved,
                 DWORD dwType,
                 const BYTE * lpData,
                cbData
);
```

参数说明如下。

hKey：要设置的键的句柄。

lpValueName：要访问的键值的名称。

lpReserved：保留值。

dwType：要设置的数据类型。

lpData：要设置的键值。

cbData：数据的长度。

4）获取键值

获取键值使用函数 RegQueryValueEX( )，其定义如下。

```
LONG RegQueryValueEx(
    HKEY hKey,
   LPSTR lpValueName,
  LPDWORD lpReserved,
  LPWORD lpType,
   LPBYTE lpData,
   LPDWORD lpcbData
  );
```

返回值：

如果函数调用成功，返回 ERROR_SUCCESS；如果函数执行失败，返回一个非零错误码。

参数说明如下。

hKey：要查询的键的句柄。

lpValueName：要查询键值的名字。

lpReserved：保留，设为 NULL。

lpType：数据缓存地址。

lpData：数据缓存大小。

5）关闭键

关闭键使用函数 RegCloseKey()，其定义如下。

```
LONG RegCloseKey(
 HKEY hKey
);
```

返回值：

如果调用成功，返回 ERROR_SUCCESS；如果函数执行失败，返回非零错误代码。

参数说明如下。

hKey：要关闭的键的句柄。

3. 应用举例——实现注册表自启动

基于上述函数，注册表自启动的实现代码如下。

```
#include <stdio.h>
#include <windows.h>
int main(void)
{
char regname[ ] = "Software//Microsoft//Windows//CurrentVersion//Run";
```

```
        HKEY hkResult;
        int ret = RegOpenKey(HKEY_LOCAL_MACHINE,regname,&hkResult);
        ret = RegSetValueEx(hkResult,"hacker"/* 注册表键名 */,0,REG_EXPAND_SZ,(unsigned char
  * )"%systemroot%//hacker.exe",25);
        if(ret == 0){
          printf("success to write run key/n");
          RegCloseKey(hkResult);
          }
        else {
        printf("failed to open regedit.%d/n",ret);
        return 0;
        }
  char modlepath[256];
        char syspath[256];
        GetModuleFileName(0,modlepath,256);      //取得程序名字
        GetSystemDirectory(syspath,256);
        ret = CopyFile(modlepath,strcat(syspath,"//hacker.exe"),1);
        if(ret)
        {
        printf("%s has been copyed to sys dir %s/n",modlepath,syspath);
        }
        else printf("%s is exisis",modlepath);
        return 0;
        }
```

2.3.6 获取网络接口信息

网络接口信息包括以下内容。

(1) 用于获取本地网络适配器信息的函数；

(2) 用于获取本地主机名、域名和 DNS 服务器信息的函数；

(3) 用于获取本地计算机网络接口数量的函数；

(4) 用于获取本地主机名、域名和 DNS 服务器信息的函数；

(5) 获取本地计算机 IP 地址表的函数。

获取网络接口信息是网络安全编程中一个比较常用的功能，例如，在数据包捕获以及数据包发送编程中，需要选择一个特定的网络接口来工作。

在 Windows 中有很多方法获取网络接口信息。例如，可以使用 WinSock 的函数 gethostbyname() 获取基本的接口信息，可以使用 WSAIoctl() 函数加标志 SIO_GET_INTERFACE_LIST 实现，还可以使用 iphlpapi 中提供的函数 GetAdaptersInfo() 获取比较详细的网络接口信息，还可以利用其他的第三方组件获取接口信息，例如 WinPcap。下面对这几种方法分别用实例说明编程实现步骤。

1. 基于 gethostbyname() 函数的方法

```
#include "stdafx.h"
#include "winsock.h"
#include <stdio.h>
```

```
#pragma comment(lib, "Ws2_32.lib")
void main()
{
WSADATA wsaData;
char HostName[255];
HOSTENT * Hostent;
int Result;
//初始化 SOCKET
Result = WSAStartup(MAKEWORD(2, 1), &wsaData);
if (Result == SOCKET_ERROR)
{
    printf("WSAStartup failed with error %d\n", Result);
    return;
}
//获取本机
Result = gethostname(HostName, 255);
printf("主机名称为: %s\n", HostName);
if (Result == SOCKET_ERROR)
{
    printf("gethostname failed with error %d\n", WSAGetLastError());
    return;
}
Hostent = (struct hostent * ) malloc(sizeof(struct hostent));
Hostent = gethostbyname(HostName);
for (int i = 0;; i++)
{
    printf("第%d个网络接口: \n", i + 1);
    printf("IP 地址: %s\n", inet_ntoa( * (IN_ADDR * )Hostent->h_addr_list[i])); //list
    if (Hostent->h_addr_list[i] + Hostent->h_length >= Hostent->h_name)
    {
        break;
    }
}
//释放 WinSock
if (WSACleanup() == SOCKET_ERROR)
{
    printf("WSACleanup failed with error %d\n", WSAGetLastError());
    return;
}
}
```

运行结果如图 2-17 所示。

2. 基于 WSAIoctl()函数的方法

在 WinSock 中使用函数 WSAIoctl()加标志 SIO_GET_INTERFACE_LIST 的方法可以获取网络接口信息。

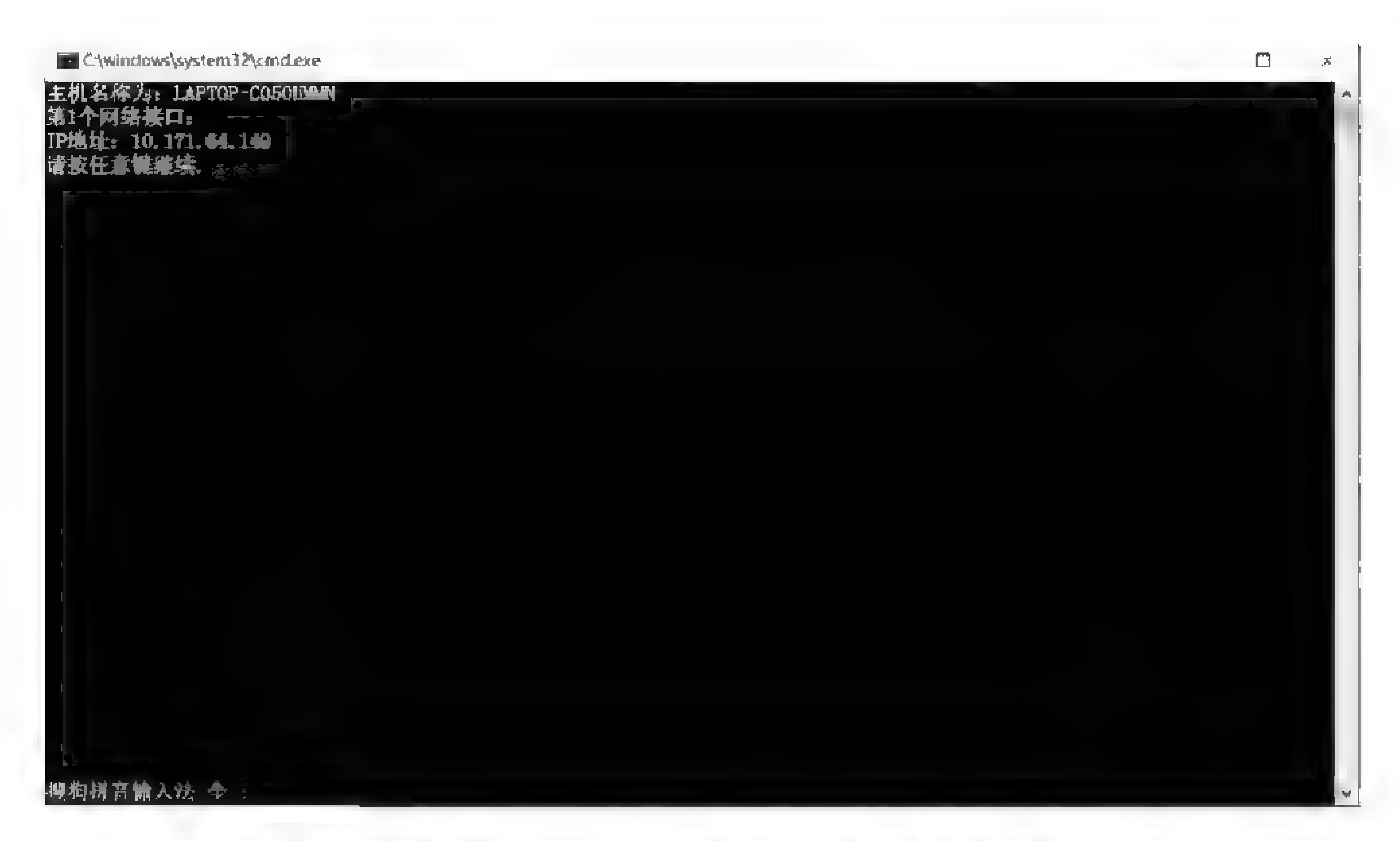


图 2-17 基于 gethostbyname()获取网络接口信息运行结果图

```
#include "stdafx.h"
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#pragma comment(lib,"WS2_32.lib")
int main()
{
WSADATA wsaData;
SOCKET socket;
//存储网络接口信息,此处设置最多存储 10 个接口信息
    INTERFACE_INFO netinterface[10];
DWORD dwByteReturned;
int InterfaceCount;
int Result;
//初始化 SOCKET
Result = WSAStartup(MAKEWORD(2, 1), &wsaData);
if(Result == SOCKET_ERROR)
{
    printf("WSAStartup failed with error %d\n", Result);
    return 1;
}
socket = WSASocket(AF_INET, SOCK_DGRAM, 0, 0, 0, 0);
if (socket == INVALID_SOCKET)
{
    printf("WSASocket failed with error %d\n\n", WSAGetLastError());
    return 1;
}
//设置标志 SIO_GET_INTERFACE_LIST
```

```
Result = WSAIoctl(socket, SIO_GET_INTERFACE_LIST,0,0, & netinterface, sizeof(netinterface),
&dwByteReturned, 0, 0);
if (Result == SOCKET_ERROR)
{
    printf("WSAIoctl failed with error %d \n", WSAGetLastError());
    return 1;
}
//读取每个网络接口信息
InterfaceCount = dwByteReturned / sizeof(INTERFACE_INFO);
printf("网络接口个数: %d\n", InterfaceCount);
for (int i = 0; i< InterfaceCount; i++) {
      char sendBuf[20] = { '\0'};
      printf("\n 第 %d 个网络接口: \n", i + 1);
       printf("IP 地址为: %s\n",inet_ntop(AF_INET,(void *)&netinterface[i].iiAddress.
AddressIn.sin_addr,sendBuf,16)); //printf("广播地址为: %s\n",inet_ntop(AF_INET, (void *)
&netinterface[i].iiBroadcastAddress.AddressIn.sin_addr, sendBuf, 16));
      printf("子网掩码为: %s\n",inet_ntop(AF_INET, (void *)&netinterface[i].iiNetmask.
AddressIn.sin_addr, sendBuf, 16));
      if (netinterface[i].iiFlags&IFF_POINTTOPOINT)
            printf("Point to Point 网络接口\n");
      if (netinterface[i].iiFlags&IFF_LOOPBACK)
            printf("回环接口\n");
      if (netinterface[i].iiFlags&IFF_BROADCAST)
            printf("支持广播\n");
      if (netinterface[i].iiFlags&IFF_MULTICAST)
            printf("支持多播\n");
      if (netinterface[i].iiFlags&IFF_UP )
      printf("接口状态: DOWN\n");
}    //释放 WinSock
if (WSACleanup() == SOCKET_ERROR)
{
      printf("WSACleanup failed with error %d\n", WSAGetLastError());
      return 1;
}
return 0;
}
```

在本程序中使用了结构类型 INTERFACE_INFO，用来存储网络接口信息，是此种方法的核心数据结构，其定义如下。

```
Typedef struct_INTERFACE_INFO{
        u_long iiFlags;
        sockaddr_gen iiAddress;
        sockaddr_gen iiBroadcastAddress;
        sockaddr_gen iiNetmask;
}INTERFACE_INFO, FAR *LPINTERFACE_INFO;
```

参数说明如下。

iiFlags：接口属性标志。其值包括：IFF_UP，表示接口正在工作；IFF_BROADCAST，表示支持广播；IFF_LOOPBACK，表示是回环接口；IFF_POINTTOPOINT，表示是 Point-

to-Point 接口；IFF_MULTICAST，表示支持多播。

iiAddress：接口的地址。

iiBroadcastAddress：广播地址。

iiNetmask：子网掩码。

运行结果如图 2-18 所示。



图 2-18　基于 WSAIoctl()函数获取网络接口信息运行结果图

3. 基于 GetAdaptersInfo()函数的方法

使用库函数 iphlpapi 中提供的函数 GetAdaptersInfo()来获取网卡信息，其函数定义的原型如下。

```
DWORD GetAdaptersInfo(
                    PIP_ADAPTER_INFO pAdapterInfo,
                    PULONG pOutBufLen
);
```

返回值：

如果函数调用成功，返回 ERROR_SUCCESS 标志。

参数说明如下。

pAdapterInfo：提供了存储网络接口的数据结构，所有得到的网络接口信息都存储在此成员中，然后就可以读取其内容获得网络接口的详细信息。

pOutBufLen：指存储网络接口信息的内存缓冲区 pAdapterInfo 的大小。如果指定的大小太小，则此函数会截取此大小信息的内容，返回 ERROR_BUFDEWR_OVERFLOW 错误标志。

实现代码如下。

```
#include "stdafx.h"
#include "winsock.h"
```

```
#include <stdio.h>
#include <iphlpapi.h>
#pragma comment(lib,"iphlpapi.lib")
void main()
{
//网络接口信息,在这里最多可以存储 20 个
IP_ADAPTER_INFO Interface[20];
PIP_ADAPTER_INFO NetInterface = NULL;
DWORD Result = 0;
unsigned long Length = sizeof(Interface);
Result = GetAdaptersInfo(Interface, &Length);
if (Result != NO_ERROR)
{
    printf("GetAdaptersInfo failed error.\n");
}
else
{
    NetInterface = Interface;
    //循环读取网络接口信息
    while (NetInterface)
    {
        static int number;
        number++;
        printf("第%d个网络接口信息:\n", number);
        printf("名称: %s\n", NetInterface->AdapterName);
        printf("信息: %s\n", NetInterface->Description);
        printf("地址: %ld\n", NetInterface->Address);
        printf("MAC 地址: %02X: %02X : %02X : %02X : %02X : %02X\n",
            NetInterface->Address[0], NetInterface->Address[1], NetInterface->Address
[2], NetInterface->Address[3],
            NetInterface->Address[4], NetInterface->Address[5]);
        printf("IP 地址: %s\n", NetInterface->IpAddressList.IpAddress.String);
        printf("IP 地址掩码: %s\n", NetInterface->IpAddressList.IpMask.String);
        printf("租用:%d\n", NetInterface->LeaseObtained);
        if (NetInterface->HaveWins)// {
            printf("Wins 配置: \n");
            printf("主 Wins 服务器: %d\n", NetInterface->PrimaryWinsServer.IpAddress.
String);     //这个用%s输出是第一个图,主 Wins 服务器后面没有任何内容,但是改成%d,第二
            //个图,后面有数字
            printf("次 Wins 服务器: %d\n", NetInterface->SecondaryWinsServer.IpAddress.
String);
        }
        else
        {
            printf("无服务器配置\n");
        }
        //读取下一个网络接口
```

```
            NetInterface = NetInterface->Next;
        }
    }
}
```

其中,成员 next 指向下一个节点,因为网络接口信息是用一个列表表示的,每个节点用来表示一个网络接口。ComboIndex 保留未用,AdapterName 表示网络接口的名称,Description 表示网卡的描述信息,AddressLength 表示网卡地址长度,Address 表示网卡地址,即 MAC 地址。Index 表示网络接口索引号,Type 表示网络接口类型,所有的网络接口类型包括:MIB_IF_TYPE_OTHER、MIB_IF_TYPE_ETHERNET、MIB_IF_TYPE_TOKENRING、MIB_IF_TYPE_FDDI、MIB_IF_TYPE_PPP、MIB_IF_TYPE_PPP、MIB_IF_TYPE_LOOPBACK、MIB_IF_TYPE_SLIP。DhcpEnabled 表示是否启用了 DHCP 服务,CurrentIpAddress 表示 IP 地址,IpAddressList 表示 IP 地址列表,GatewayList 表示网关地址列表,DhcpServer 表示 DHCP 服务地址,HaveWIns 表示是否启动了 WINS 服务,PrimaryWinsServer 表示主 WINS 服务地址,SecondaryWinsServer 表示次 WINS 服务地址,LeaseObtaine 表示 DHCP 租赁时间,LeaseExpires 表示 DHCP 租赁失效时间。

注意:函数 GetAdaptersInfo( )不返回回环网络接口信息。另外,在新的 Windows 操作系统中建议使用函数 GetAdaptersAddresses( )。

运行结果如图 2-19 所示。



图 2-19　基于 GetAdaptersInfo( )函数获取网络接口信息运行结果图

4. 基于 WinPcap 的方法

使用 WinPcap 中提供的函数也能获取网络接口信息,而且更加方便实用。WinPcap 是主要提供网络数据包捕获功能的网络组建,它提供了各种网络功能函数,其中函数 pcap_findalldevs()提供了获取网络接口信息的功能。使用 WinPcap 可以方便地获取网络接口信

息，它是由函数 pcap_findalldevs()来完成的。其定义如下。

```
int pcap_findalldevs (pcap_if_t * . * alldevsp, Char * errbuf )
```

此函数获取的网络接口信息，把它放在一个网络接口列表 alldevsp 里面，它存储了网络接口的基本信息，包括其名称、详细说明以及 IP 地址和掩码信息等内容。其数据结构 pcap_if 定义如下。

```
struct pcap_if {  struct pcap_if * next;
                  char * name;
                  char * descriptions;
                  struct pcap_addr * addresses;
                  u_int flags;
                }
```

返回值：

函数执行失败，返回－1，参数 errbuf 返回出错信息，成功执行返回 0。

参数说明如下。

next：下一个列表节点。

name：网络接口名称。

descriptions：网络接口的描述信息。

addresses：网络接口的 IP 地址列表。

flags：标志，可用的标志有 PCAP_IF_LOOPBACK，表示是否是回环网络接口。

注意：在 WinPcap 最新的版本中，另外提供了函数 pcap_findalldevs_ex( )用来捕获网络接口信息。

实现代码如下。

```
#include "stdafx.h"
#include "stdio.h"
#include "pcap.h"
#pragma comment(lib,"wpcap.lib")
#pragma comment(lib, "ws2_32.lib")

//使用 WinPcap 获取网络接口信息
int main()
{
int Result, i;
char PcapError[PCAP_ERRBUF_SIZE];
pcap_if_t * Interface;
pcap_if_t * NetInterface;
Result = pcap_findalldevs(&NetInterface, PcapError);
if (Result == -1 || NetInterface == NULL)
    return FALSE;
for (Interface = NetInterface, i = 0; Interface&&i < 10; Interface = Interface->next, i++)
{
    pcap_addr_t * a;
    printf("第 % d 个网络接口:\n", i++);
```

```
    printf("名称: % s\n", Interface->name);     //网络接口名称
                                                //网络接口信息
    if (Interface->description)
        printf("信息: % s\n", Interface->description);
    //地址信息
    for (a = Interface->addresses; a; a = a->next)
    {
        char sendBuf[20] = { '\0'};
        switch (a->addr->sa_family)
        {
        case AF_INET:
            if (a->addr)
                  printf("IP 地址: % s\n",
                        inet_ntop(AF_INET,(void * )&((struct sockaddr_in * )a->addr)->
sin_addr,sendBuf,16));
            if (a->netmask)
                  printf("地址掩码: % s\n",
                        inet_ntop(AF_INET,(void * )&((struct sockaddr_in * )a->netmask)->
sin_addr,sendBuf,16));
            break;
        default:
            break;
        }
    }
}
pcap_freealldevs(NetInterface);
return 1;
}
```

运行结果如图 2-20 所示。



图 2-20 基于 WinPcap 获取网络接口信息运行结果图

小　结

本章介绍了套接字编程的概念、连接过程、基本套接字以及典型的过程图，在此基础上利用 WinSock 编程相关函数实现套接字通信。然后，针对 Visual C++的网络安全编程介绍了获取系统实时信息、进程处理、线程处理、定时器处理、获取网络接口信息的具体方法。

思 考 题

1. 什么是端口？网络通信中为什么要引入端口？
2. 什么是套接字？
3. 套接字有哪些种类？分别有什么特点？
4. 什么是网络编程？简述应用进程间的两种通信方式。
5. 简述客户/服务器的通信过程。
6. 进程与线程的区别是什么？
7. 网络接口信息包括哪些？
8. 编程模拟实现 Windows 平台下两个进程 a 与 b 之间的通信过程，要求实现消息的发送和接收。
9. 创建收发文件的服务器端、客户端，编写程序实现如下功能。

(1) 客户端接收用户输入的传输文件名；

(2) 客户端请求服务器端传输该文件名所指文件；

(3) 如果指定文件存在，服务器端就将其发给客户端，否则断开连接。

10. 编程实现一个定时器：每隔 1s 输出一个英文字符 n。

第3章　密码学编程

密码学是信息安全的基础，可应用于数据加解密、数字签名、安全认证、电子投票等领域。在理解密码学基本概念的基础上，可应用经典密码算法解决实际系统中的安全问题。

3.1　密码学基本概念

经典密码学是指秘密书写的科学。密码(cipher)是一种秘密书写的方法。把明文(plaintext)变换为密文(ciphertext)或密报(cryptography)，这种变换叫加密(encipherment或encryption)。而将密文变换为明文的过程称为解密(decipherment或decryption)。

加密和解密都要通过密钥(Key)的控制。加密/解密示意图如图3-1所示。

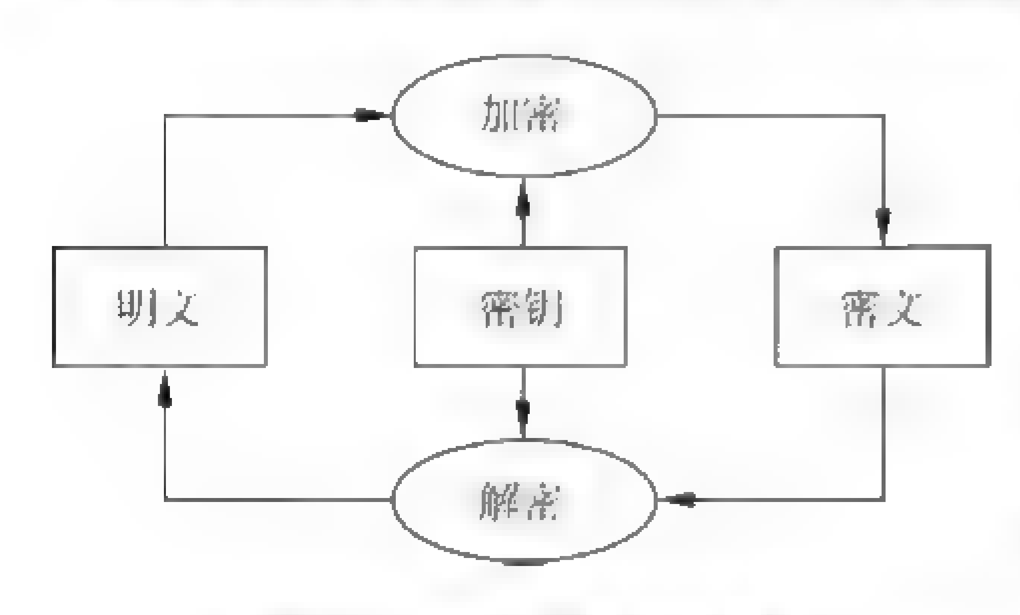


图3-1　加密/解密示意图

密码学的研究领域可分为密码编码学(Cryptography)和密码分析学(Cryptanalysis)两个分支，分别研究密码的编制和破译问题。密码编码学和密码分析学是密码学的两个方面，两者既相互对立，又互相促进和发展。

密码编码学研究密码编码(也称为加密)、译码(也称为解密)的理论和算法。密码分析也叫做破译，密码分析学的主要任务是研究加密信息的破译或认证信息的伪造。它主要是对密码信息的解析方法进行研究。只有密码分析者才能评判密码体制的安全性。

3.1.1　对称密码

对称密码算法又称为传统密码算法，其主要特征是加密算法与解密算法所使用的密钥是相同的，或者从一个容易推出另一个。对称密码算法可用于保护数据的机密性和完整性，还可以扩展到身份识别等。对称密码算法在最近半个多世纪的研究中得到了迅猛发展，有很多成熟的算法可供选择。具有代表性意义的算法有两类：一类是分组密码算法；另一类是序列密码算法。分组密码算法是把明文、密文分成等长的组，然后对这些等长的组进行变换，把明文变为密文，把密文变为明文。而序列密码算法则是通过算法把密钥k扩展为与明文或密文相一致的子密钥序列，然后明文与密文通过与该子密钥序列按位模2相加，把明文变为密文，把密文变为明文。代表性分组密码算法有DES、AES。其他的对称分组密码算法包括IDEA，RC5，CAST-128等。

3.1.2　公钥密码

公钥密码算法又称为非对称密钥密码算法，其主要特征是加密密钥可以公开，而不会影

响到解密密钥的机密性。它可用于保护数据的机密性、完整性和身份识别等。

公钥密码体制也称为双密钥密码体制或非对称密码体制，与此相对应，将序列密码和分组密码等称为单密钥密码体制或对称密钥密码体制。表 3-1 总结了单钥加密和公开密钥加密的重要特征。

表 3-1　单钥加密与公钥加密

选项	单钥加密	公开密钥加密
运行条件	① 加密和解密使用同一密钥和同一算法 ② 发送方和接收方必须共享密钥和算法	① 用同一算法进行加密和解密，而密钥有一对，其中一个用于加密，而另一个用于解密 ② 发送方和接收方每个拥有一个相互匹配的密钥中的一个(不是另一个)
安全条件	① 密钥必须保密 ② 如果不掌握其他信息，要想解密报文是不可能或者至少是不现实的 ③ 知道所用的算法加上密文的样本必须是不足以确定密钥的	① 两个密钥中的一个必须保密 ② 如果不掌握其他信息，要想解密报文是不可能或者至少是不现实的 ③ 知道所用的算法加上一个密钥和密文的样本必须不足以确定密钥

为了区分这两个体制，一般将单钥加密中使用的密钥称为秘密密钥(Secret Key)，公开密钥加密中使用的两个密钥分别称为公开密钥(Public Key)和私有密钥(Private Key)。在任何时候私有密钥都是保密的，但把它称为私有密钥而不是秘密密钥，以免同单钥加密中的秘密密钥混淆。

单钥密码安全的核心是通信双方秘密密钥的建立，当用户数增加时，其密钥分发就越来越困难，而且单钥密码不能满足日益膨胀的数字签名的需要。公开密钥密码编码学是在试图解决单钥加密面临的这个难题的过程中发展起来的。公共密钥密码的优点是不需要经安全渠道传递密钥，大大简化了密钥管理。它的算法有时也称为公开密钥算法或简称为公钥算法。公开密钥的应用主要有以下三方面。

(1) 加密和解密。发送方用接收方的公开密钥加密报文。

(2) 数字签名。发送方用自己的私有密钥"签署"报文。签署功能是通过对报文或者作为报文的一个函数的一小块数据应用发送者私有密钥加密完成的。

(3) 密钥交换。两方合作以便交换会话密钥。

典型的公钥密码算法包括：RSA 算法，有限域乘法群密码与椭圆曲线密码。

3.1.3　哈希函数

1. 哈希函数的概念及应用

哈希函数是为了实现数字签名或计算消息的鉴别码而设计的。哈希函数以任意长度的消息作为输入，输入一个固定长度的二进制值，称为哈希值、杂凑值或消息摘要。从数学上看，哈希函数 H 是一个映射：

$$H: Z_2^* \to Z_2^n$$

$$x \to H(x)$$

这里，n 是一个给定的自然数，称为杂凑长度。用 Z_2^m 表示长度为 $m$bit 的全体二进制数的集合，而 $Z_2^* = \bigcup_{m \in N} Z_2^m$。

单向哈希函数或者安全哈希函数对于消息鉴别和数字签名都是很重要的。

2. 哈希函数的要求

(1) H 可应用于任意大小的数据块。

(2) H 产生一个固定长度的输出。

(3) 对于任意给定的 x,计算 $H(x)$ 比较容易,用硬件和软件均可实现。

(4) 对于任意给定的散列码 h,找到满足 $H(x)=h$ 的 x 在计算上是不可行的。具有这种性质的散列函数称为单向或抗原像的。

(5) 对于任意给定的分组 x,找到满足 $y<>x$ 且 $H(y)=H(x)$ 的 y 在计算上是不可行的。具有这种性质的散列函数称为抗第二原像的,有时也被称为弱抗碰撞的。

(6) 找到任何满足 $H(y)=H(x)$ 的偶对 (x,y) 在计算上是不可行的。具有这种性质的散列函数称为抗碰撞的,有时也称为强抗碰撞的。

3. 哈希函数的安全性

与对称加密一样,对安全哈希函数的攻击也有两种方法:密码分析和强力攻击。与对称加密算法一样,哈希函数的密码分析设计利用算法的逻辑弱点。

4. 哈希函数的应用

消息认证:消息认证是用来验证消息完整性的一种机制或服务。消息认证确保收到的数据确实和发送时一样(即没有修改、插入、删除或重放)。此外,通常还要求消息认证机制确保发送方声称的身份是真实有效的。消息认证中使用哈希函数的本质如下:发送者根据待发送的消息使用该函数计算一组哈希值,然后将哈希值和消息一起发送出去。接收者收到后对于消息执行同样的哈希计算,并将结果与收到的哈希值进行对比。如果不匹配,则接收者推断出消息遭受了篡改。

口令:在基于口令的身份认证机制中,操作系统存储的是口令的哈希值,而不是口令本身,因此,获得口令文件访问的黑客并不能获取实际的口令。简单来说,当用户输入口令时,该口令的哈希值与存储在系统中的哈希值进行比较验证。这个应用要求哈希函数具有抗原像性,或许还要求抗第二原像。

入侵检测和病毒检测:在系统中为每个文件存储 $H(F)$ 并保护好该哈希值(例如,存储在一个受保护的 CD-R 上)。可以通过重新计算 $H(F)$ 确定文件是否已被修改。入侵者可能会改变 F,但不可能改变 $H(F)$。这个应用要求哈希函数抗第二原像。

密码学哈希函数也能够用于构建随机函数(PRF)或用作伪随机数发生器(PRNG)。

3.1.4 数字签名

对文件进行加密只解决了传送信息的保密问题,而防止他人对传输的文件进行破坏以及如何确定发信人的身份还需要采取其他的手段,这一手段就是数字签名。在信息安全保密系统中,数字签名技术有着特别重要的地位,信息安全服务中的源鉴别、完整性服务、不可否认服务中,都要用到数字签名技术。

数字签名算法属于公钥密码范畴,它的签名密钥是私钥,验证密钥是公钥。主要用途是完成数字签名,从而实现抵抗赖、消息鉴别和身份识别。它与公钥加密算法的公私钥生成算法是一样的,区别只是在于:公钥加密算法使用公钥进行加密,用私钥进行解密;而签名算

法中公钥和私钥的角色是对换了的，它使用私钥进行加密，公钥进行解密也即验证。

数字签名是一种类似写在纸上的普通的物理签名，但是使用了私钥加密领域的技术实现，用于鉴别数字信息的方法。一套数字签名通常定义两种互补的运算，一个用于签名，另一个用于验证。数字签字由公钥密码发展而来，它在网络安全，包括身份认证、数据完整性、不可否认性以及匿名性等方面有着重要应用。特别是在大型网络安全通信中的密钥分配、认证以及电子商务系统中都有重要的作用，数字签名的安全性日益受到高度重视。

为了保证信息的完整性与真实性，数字签名技术必须具有以下三个基本功能：发送者不能抵赖对消息的签名、接收者能够核实发送者对消息的签名，接收者不能伪造对方的签名。换句话说，数字签名技术必须具有普通签名的特点。

而数字签名的特点是它代表了消息的特征，消息如果发生改变，数字签名的值也将发生改变，不同的消息将得到不同的数字签名。安全的数字签名使接收方可以得到保证：消息确实来自发送方。因为签名的私钥只有发送方自己保存，他人无法做一样的数字签名，如果第三方冒充发送方发出一个消息，而接收方在对数字签名进行解密时使用的是发送方的公开密钥，只要第三方不知道发送方的私有密钥，加密出来的数字签名和经过计算的数字签名必然是不相同的，这就提供了一个安全的确认发送方身份的方法，即数字签名的真实性得到了保证。

假定 A 给 B 发送了一个带签名的消息 M，则 A 的数字签名必须满足下述条件。

(1) B 能够证实 A 对消息 M 的签名；

(2) 任何人，包括 B，都不能伪造 A 对 M 的签名；

(3) 如果 A 否认了 B 对 M 的签名，可以通过仲裁机构解决 A 和 B 的争议。

数字签名类似手书签名，它具有以下的性质。

(1) 能够验证签名产生者的身份，以及产生签名的日期和时间；

(2) 能用于证实被签名消息内容；

(3) 数字签名可由第三方验证，从而能够解决通信双方的争议。

为了实现数字签名的以上性质，它就应满足下列要求。

(1) 签名是可信的：任何人都可以验证签名的有效性。

(2) 签名是不可伪造的：除了合法的签名者外，任何人伪造其签名是困难的。

(3) 签名是不可复制的：对一个消息的签名不能通过复制变为另一个消息的签名，如果一个消息的签名是从别处复制得到的，则任何人都可以发现消息与签名之间的不一致性，从而可以拒绝签名的消息。

(4) 签名的消息是不可改变的：经签名的消息不能篡改，一旦签名的消息被篡改，任何人都可以发现消息与签名之间的不一致性。

(5) 签名是不可抵赖的：签名者事后不能否认自己的签名。可以由第三方或仲裁方来确认双方的信息，以做出仲裁。

为了满足数字签名的这些要求，例如，通信双方在发送消息时，既要防止接收方或其他第三方伪造，又要防止发送方因对自己的不利而否认，也就是说，要保证数字签名的真实性。

3.1.5　随机数与伪随机数

1. 随机数的使用

许多基于密码学的网络安全算法使用了随机数。例如：

(1) RSA 公钥加密算法和其他公钥算法中的密钥生成；

(2) 对称流密码的密钥流生成；

(3) 用作临时会话密钥或创建数字信封的对称密钥的生成；

(4) 在许多密钥分配方案中，例如 KERBEROS，使用随机数进行握手以防止重放攻击；

(5) 会话密钥的生成，无论是通过密钥分配还是由主体之一完成。

这些应用对随机数提出了两种截然不同的且不能互相兼容的要求：随机性和不可预测性。

随机性：传统上，对所谓随机数字序列的生成的关注一直在与数字序列是否具有某些明确定义的统计意义上的随机性。

验证数字序列随机性的准则如下。

均匀分布性：数字序列的分布应是均匀的，即每个数的出现频率大致相等。

独立性：序列中的任何数不能由其他数推导出来。

2. 随机与伪随机

密码应用通常使用算法生成随机数。这些算法是确定性的，因此无法产生统计随机的数字序列。但是，如果该算法是良好的，所得到的序列将能经受住许多合理的随机性测试，这样的数字被称为伪随机数。

3.2 基于 SHA-1 算法的文件完整性校验

该算法实现如下功能。

(1) 编写应用程序，正确实现 SHA-1 算法。

(2) 程序不仅能够为任意长度的字符串生成 SHA-1 摘要，而且可以为任意大小的文件生成 SHA-1 摘要。

(3) 程序还可以利用 SHA-1 摘要验证文件的完整性。验证文件的完整性有两种方式：一种是手动输入 SHA-1 摘要的条件下，计算出当前被测文件的 SHA-1 摘要，再将两者进行比对；另一种是先利用系统工具 SHA-1sum 为被测文件生成一个.SHA-1 的同名文件，然后让程序计算出被测文件的 SHA-1 摘要，将其余与.SHA-1 文件中的 SHA-1 摘要进行比较，最后得出检测结果。具体要求有以下几点。

1. 程序的输入格式

程序为命令行程序，可执行文件名为 SHA-1.exe，命令行格式如下。

```
./SHA-1 [选项][被测文件路径][.SHA-1 文件路径]
```

其中，[选项]是程序为用户提供的各种功能。在本程序中，[选项]包括{-h，-t，-c，-v，-f}5 个基本功能。[被测文件路径]为应用程序指明被测文件所在文件系统中的路径。[.SHA-1 文件路径]为应用程序指明由被测文件生成的。其中第一个参数为必选项，后两个参数可以根据功能进行选择。

2. 程序的执行过程

(1) 打印帮助信息；

(2) 在控制台命令行中输入./ -h打印程序的帮助信息；

(3) 打印测试信息；

(4) 在控制台命令行中输入./SHA-1 -t打印程序的测试信息；

(5) 为指定文件生成SHA-1摘要；

(6) 在控制台命令行中输入./SHA-1 -c[被测文件路径]，计算出被测文件的SHA-1摘要并打印出来。

3. 验证文件完整性方法1

在控制台命令行中输入./SHA-1 -c[被测文件路径]，程序会先让用户输入被测文件的SHA-1摘要，然后再重新计算被测文件的SHA-1摘要，最后将两个摘要逐位比较。若一致，则说明文件是完整的，否则，说明文件遭到了破坏。

4. 验证文件完整性方法2

在控制台命令行中输入./SHA-1 -f [被测文件路径][.SHA-1文件路径]，程序会自动读取.SHA-1文件中的摘要，然后再重新计算出被测文件的SHA-1摘要，最后将两者逐位比较。若一致，则说明文件是完整的，否则，说明文件遭到了破坏。

3.2.1　SHA-1算法

1. 安全散列函数算法(SHA)

近年来，一直使用广泛的散列函数是安全散列算法(Secure Hash Algorithm，SHA)。SHA由美国国家标准与技术研究院(National Institute of Standards and Technology，NIST)设计，并于1993年作为美国联邦信息处理标准(FIPS180)发布。当SHA的缺点被发现后，1995年发布了修订版FIPS180-1，通常称之为SHA-1。SHA-1产生160位的散列值。2002年，NIST提出了该标准的修订版FIPS180-2，它定义了3种新的SHA版本，散列长度分别为256位、384位和512位，分别称为SHA-256、SHA-384及SHA-512。这些新版本具有与SHA-1相同的基础结构，并使用相同类型的模算术运算和逻辑二进制运算。

研究人员已经证明SHA-1的安全性弱于它的160位散列长度，有必要转向使用较新版本的SHA。

2. SHA-1算法原理

SHA-1算法由NIST与美国国家安全局设计，并且被美国政府采纳，成为美国国家标准。事实上，SHA-1目前是全世界使用最为广泛的哈希算法，已经成为业界的事实标准。可以对长度不超过2^{64}b的消息进行计算，输入以512位数据块为单位处理，产生160b长的消息摘要作为输出。

1) 数据填充与分拆

在SAH-1中，对于输入的任意长度的消息X，先把它扩充成长度(位数)为512的整倍数的数据：

$$X \to X \quad \| 1 \| 0 \cdots 0 \| \quad (X\text{的长度}) \quad L$$

(原消息)　(填充)　(64b)

再把所得数据分成s个512b长的数组：

$$X = x_1 \| x_2 \| \cdots \| x_s$$

2) SHA 算法描述

(1) SHA 的初始化和主循环

SHA-1 有 5 个 32b 的链接变量 A,B,C,D,E。算法执行时对 A,B,C,D,E 初始化为(十六进制表示)：

$$A=0x67452301$$
$$B=0xefcdab89$$
$$C=0x98badcfe$$
$$D=0x10325476$$
$$E=0xc3d2e1f0$$

如图 3-2 所示给出了 SHA-1 的主循环结构图。它执行 s 次循环，把链接变量的初始值，在逐次循环中变换，产生最终的哈希值。每个主循环都由 4 个轮循环组成，每轮 20 次操作，每次操作对 a、b、c、d、e 中的三个进行一次非线性运算，后进行移位和加运算。a、b、c、d 和 e 分别加上 A、B、C、D 和 E，然后用下一数据分组继续运行算法。最后的输出由 A、B、C、D 和 E 级联而成。

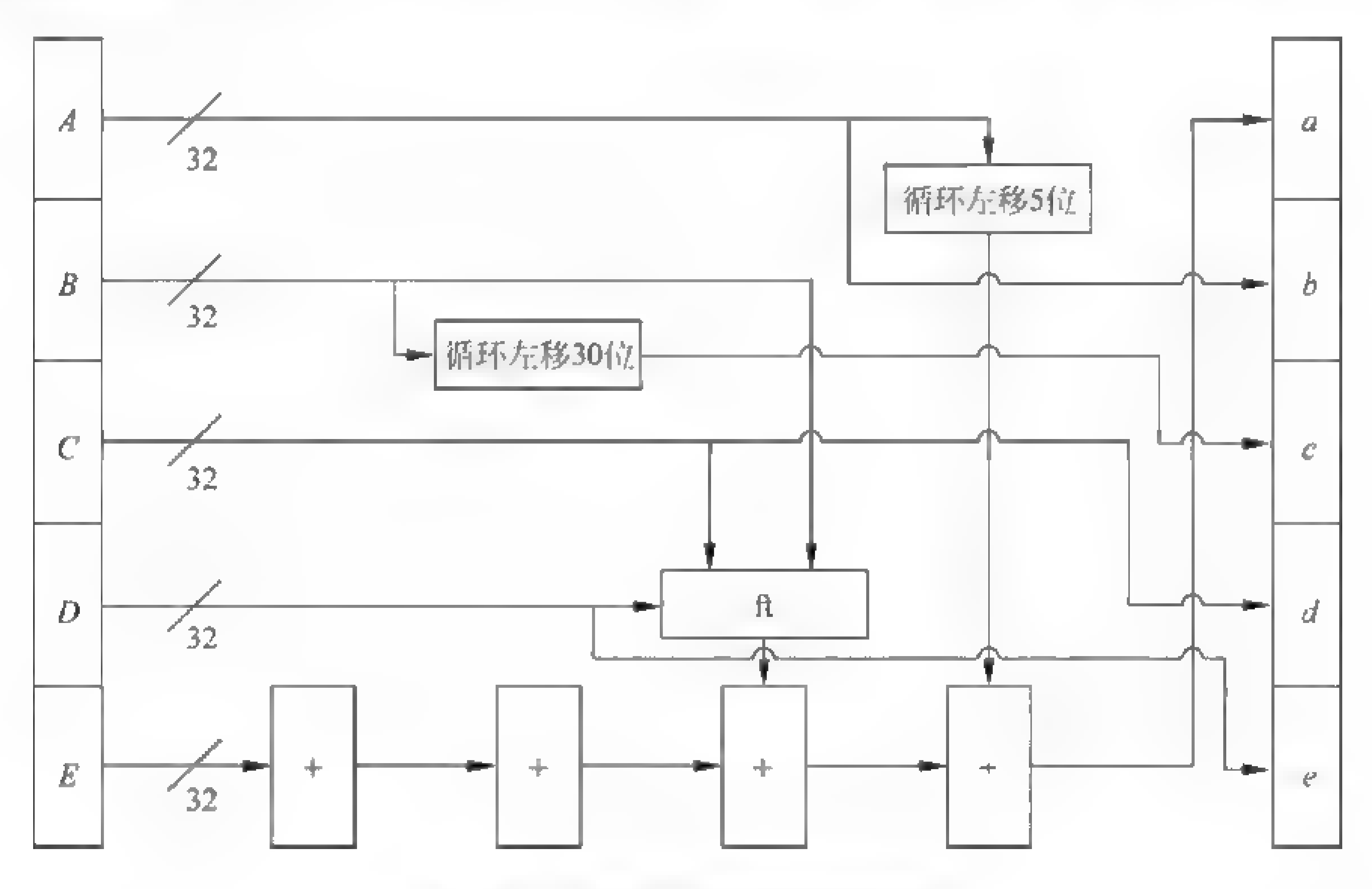


图 3-2 SHA-1 的主循环结构

(2) 轮函数

SHA-1 的 4 个轮函数中的每一轮都由 20 次的操作组成，4 轮共完成 80 次操作。SHA-1 中定义了三个基本逻辑函数，它们合并为一个带参数 i(表示操作序号)的逻辑函数，用在 4 轮的 80 个操作中。设 X,Y,Z 表示 32b 的字，定义如下：

$$(X,Y,Z)=\begin{cases}(X\wedge Y)\vee(X\wedge Z) & 0\leqslant i\leqslant 19\\ X\oplus Y\oplus Z & 20\leqslant i\leqslant 39,\quad 60\leqslant i\leqslant 79\\ (X\wedge Y)\vee(X\wedge Z)\vee(X\wedge Z) & 40\leqslant i\leqslant 59\end{cases}$$

各个轮函数的输入除了链接变量外，另一个输入是 512b 的字分组的扩展。若把这 16 个 32b 字分组表示，先把它扩展为 80 次操作中的所需要的 80 个 32b 如下：

$$\begin{cases} M_t & 0 \leqslant i \leqslant 15 \\ (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}) <<< 1 & 16 \ll i \ll 79 \end{cases}$$

轮函数中还有 4 个常数。按照 80 次操作，它们记为：

$$K_t = \begin{cases} \text{0x5a827999} & 0 \leqslant i \leqslant 19 \\ \text{0x6ed9eba1} & 20 \leqslant i \leqslant 39 \\ \text{0x8f1bbcdc} & 40 \leqslant i \leqslant 59 \\ \text{0xca62c1d6} & 60 \leqslant i \leqslant 79 \end{cases}$$

现在已为每个操作准备了逻辑函数、32b 消息字和轮常量。这里 i 对应操作序号(79)，表示循环左移 sb 运算，$\oplus$表示模加法。

这时，主循环可以表示如下：

$$a = A,\quad b = B,\quad c = C,\quad d = D,\quad e = E$$

对 $i=0$ to 79 执行

$$\text{TEMP} = (a <<< 5) + f_i(b,c,d) + e + W_i + K_t$$
$$e = d$$
$$d = c$$
$$c = b <<< 30$$
$$b = a$$
$$a = \text{TEMP}$$

80 次循环后，计算 $A=a+A, B=b+B, C=c+C, D=d+D, E=e+E$。

然后，利用下一次 512b 分组进行计算，直至用完最后一个 512b 分组为止。这时变量 A,B,C,D,E 的当前值的毗连：$A \| B \| C \| D \| E$，即是所要的 Hash 值。

SHA-1 算法实现流程如图 3-3 所示。

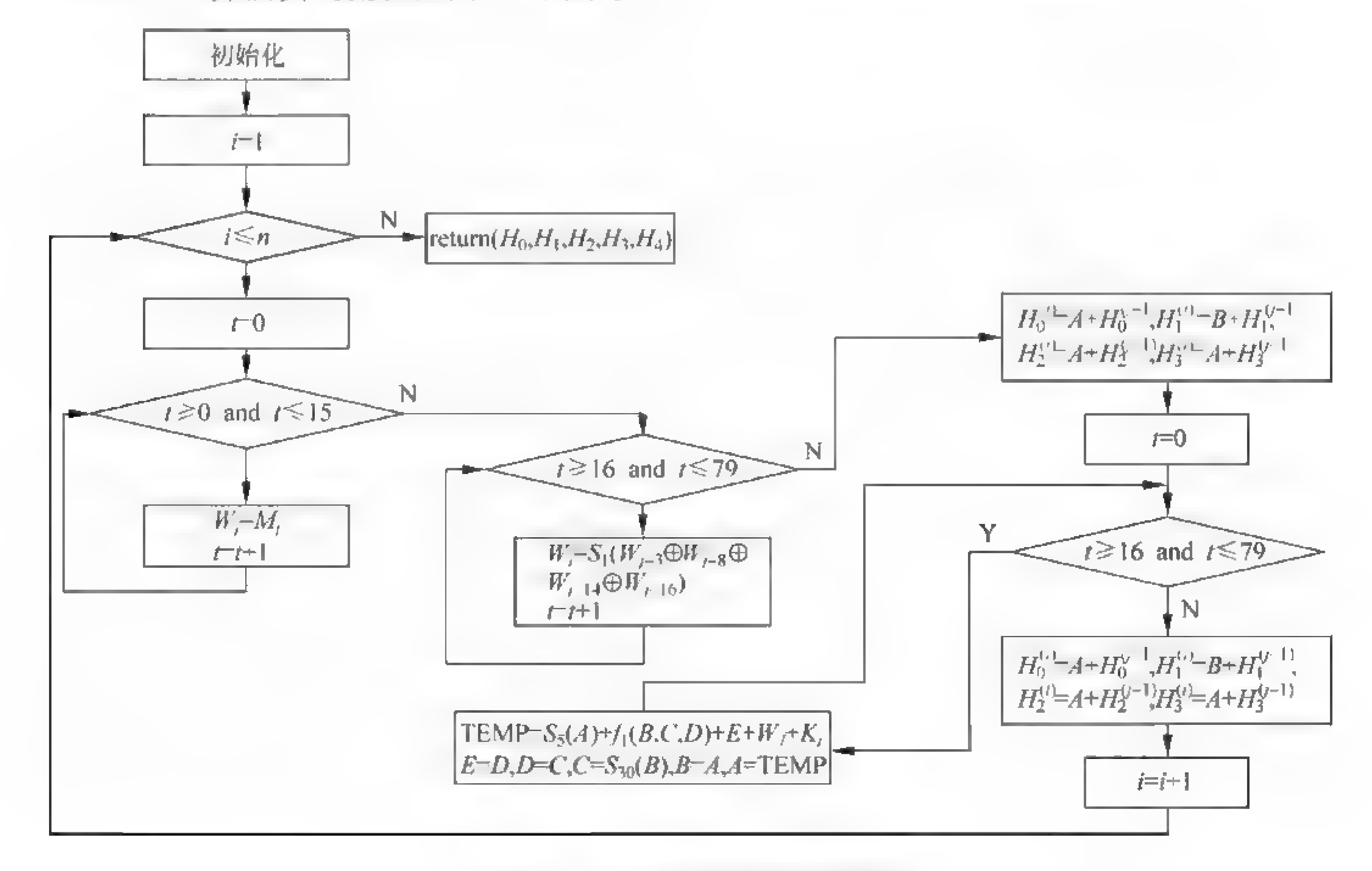


图 3-3　SHA-1 算法的整体流程图

3) SHA-1 算法实现

```
//SHA-1.cpp : 定义控制面板应用程序的入口点
#include "stdafx.h"
#include<stdio.h>
#include<string.h>
#include<conio.h>
#include<wtypes.h>
void creat_w(unsigned char input[64], unsigned long w[80])
{
int i, j; unsigned long temp, temp1;
for (i = 0; i<16; i++)
{
    j = 4 * i;
    w[i] = ((long)input[j]) << 24 | ((long)input[1 + j]) << 16 | ((long)input[2 + j]) << 8 |
((long)input[3 + j]) << 0;
}
for (i = 16; i<80; i++)
{
    w[i] = w[i - 16] ^ w[i - 14] ^ w[i - 8] ^ w[i - 3];
    temp = w[i] << 1;
    temp1 = w[i] >> 31;
    w[i] = temp | temp1;
}
}
char ms_len(long a, char intput[64])
{
unsigned long temp3, p1; int i, j;
temp3 = 0;
p1 = ~(~temp3 << 8);
for (i = 0; i<4; i++)
{
    j = 8 * i;
    intput[63 - i] = (char)((a&(p1 << j)) >> j);
}
return '0';
}
int _tmain(int argc, _TCHAR * argv[])
{
unsigned long H0 = 0x67452301, H1 = 0xefcdab89, H2 = 0x98badcfe, H3 = 0x10325476,
H4 = 0xc3d2e1f0;
unsigned long A, B, C, D, E, temp, temp1, temp2, temp3, k, f; int i, flag; unsigned long w[80];
unsigned char input[64]; long x; int n;
printf("input message:\n");
scanf("%s", input);
n = strlen((LPSTR)input);
if (n<57)
{
    x = n * 8;
    ms_len(x, (char * )input);
```

```
        if (n == 56)
            for (i = n; i<60; i++)
                input[i] = 0;
        else
        {
            input[n] = 128;
            for (i = n + 1; i<60; i++)
                input[i] = 0;
        }
    }
    creat_w(input, w);
    /* for(i=0;i<80;i++)
    printf("%lx,",w[i]); */
    printf("\n");
    A = H0; B = H1; C = H2; D = H3; E = H4;
    for (i = 0; i<80; i++)
    {
        flag = i / 20;
        switch (flag)
        {
        case 0: k = 0x5a827999; f = (B&C) | (~B&D); break;
        case 1: k = 0x6ed9eba1; f = B^C^D; break;
        case 2: k = 0x8f1bbcdc; f = (B&C) | (B&D) | (C&D); break;
        case 3: k = 0xca62c1d6; f = B^C^D; break;
        }
        /* printf("%lx,%lx\n",k,f); */
        temp1 = A << 5;
        temp2 = A >> 27;
        temp3 = temp1 | temp2;
        temp = temp3 + f + E + w[i] + k;
        E = D;
        D = C;
        temp1 = B << 30;
        temp2 = B >> 2;
        C = temp1 | temp2;
        B = A;
        A = temp;
        //printf("%lx,%lx,%lx,%lx,%lx\n",A,B,C,D,E);     //输出编码过程
    }
    H0 = H0 + A;
    H1 = H1 + B;
    H2 = H2 + C;
    H3 = H3 + D;
    H4 = H4 + E;
    printf("\noutput hash value:\n");
    printf("%lx%lx%lx%lx%lx", H0, H1, H2, H3, H4);
    getch();
    }
```

运行结果如图 3-4 所示。



图 3-4　基于 SHA 计算消息摘要运行结果图

3.2.2　基于 SHA-1 的文件完整性检验

1. 基本步骤

文件完整性检验在 main()函数中实现。应用程序为用户提供了多个选项，不但可以在命令行下计算文件的 SHA-1 摘要，验证文件的完整性，还可以显示程序的帮助信息和 SHA-1 算法的测试信息。

在 main 函数中，程序通过区分参数 argv[1]的不同值来启动不同的工作流程。如果 argv[1]等于"-h"，表示显示帮助信息；如果 argv[1]等于"-t"，表示显示测试信息；如果 argv[1]等于"-c"，表示计算被测文件的 SHA-1 摘要；如果 argv[1]等于"-v"，表示根据手工输入的 SHA-1 摘要验证文件的完整性；如果 argv[1]等于"-h"，表示根据.SHA-1 文件中的摘要验证文件的完整性。

帮助信息可以协助用户快速地了解命令行输入格式。测试信息可以让用户验证 SHA-1 算法的正确性。用于测试的消息字符串都是 SHA-1 算法官方文档(RFC1321)中给出的例子，如果计算的结果相同，则说明程序的 SHA-1 运算过程正确无误。

程序提供了两种验证文件完整性的方式：一种是让用户手工输入被测文件的 SHA-1 摘要，然后调用 SHA-1 类的运算函数重新计算被测文件的 SHA-1 摘要，最后将两个摘要逐位进行比较，进而验证文件的完整性；另一种是从与被测文件对应的 SHA-1 文件中读取 SHA-1 摘要，然后调用 SHA-1 类的运算函数重新计算被测文件的 SHA-1 摘要，最后将两个摘要逐位进行比较，进而验证文件的完整性。

手工输入验证分为以下 6 个步骤。

(1) 首先比较参数 argv[1]，判断是否通过手工输入进行验证。若是，则继续下面的步骤；否则，退出。

(2) 检测被测文件的路径是否存在，若存在，则继续下面的步骤；否则，退出。

(3) 打开被测文件的 SHA-1 摘要并保存在数组 InputSHA-1 中。

(4) 打开被测文件，读取被测文件的内容，并调用 Update 函数重新计算被测文件的 SHA-1 摘要。

(5) 调用 Tostring 函数将 SHA-1 摘要表示成十六进制字符串的形式。

(6) 最后调用 strcmp 函数判断两个摘要是否相同，若相同，则说明被测文件是完整的；否则，说明文件受到了破坏。

2. 实现代码

```
//头文件
#include "SHA-1.h"
```

```
#include <iostream>
using namespace std;
//功能函数
//Main 函数
//
int main(int argc,char * argv[])
{
   char * pFilePath;                    //需要进行 SHA-1 计算的文件路径
   char * pSHA-1FilePath;               //存放 SHA-1 摘要的 SHA-1 文件路径
   char SHA-1Digest[33];                //SHA-1 摘要,用于存放手动输入的 SHA-1 摘要信息
   char SHA-1Record[50];                //SHA-1 文件中的一行记录
   string strTmp;                       //字符串定义
       char * pHelpMsg = {"-h"};        //帮助信息
       char * pTestMsg = {"-t"};        //SHA-1 测试程序的应用信息
       char * pCompute = {"-c"};        //计算指定文件的 SHA-1 摘要
       char * pMValidate = {"-mv"};     //手动对文件进行 SHA-1 认证
       char * pfValidate = {"-fv"};     //通过比较对文件的 SHA-1 摘要进行认证
       char * pSpace = {" "};           //定义空格

       //参数检测
       if(argc<2 || argc>4)
       {
            cout <<"Parameter Error !"<< endl;
            return -1;
       }
     //显示帮助信息
        if((argc == 2)&&(!strcmp(pHelpMsg,argv[1])))
        {
             cout <<"SHA-1 usage: [-h] -- help information"<< endl;
             cout <<" [-t] -- test SHA-1 application"<< endl;
             cout <<" [-c] [file path of the file computed]"<< endl;
             cout <<" -- compute SHA-1 of the given file"<< endl;
             cout <<" [-mv] [file path of the file validated]"<< endl;
              cout <<" -- validate the integrality of a given file by manual input SHA-1 
value"<< endl;
cout <<" [-fv] [file path of the file validated] [file path of the .SHA-1 file]"<< endl;
             cout <<" -- validate the integrality of a given file by read SHA-1 value from 
.SHA-1 file"<< endl;
        }
        //显示 SHA-1 应用程序的测试信息
  if((argc == 2)&&(!strcmp(pTestMsg,argv[1])))
        {
             cout <<"SHA-1(\"\") = "<< SHA-1("").toString()<< endl;
             cout <<"SHA-1(\"a\") = "<< SHA-1("a").toString()<< endl;
             cout <<"SHA-1(\"abc\") = "<< SHA-1("abc").toString()<< endl;
             cout <<"SHA-1(\"message digest\") = "<< SHA-1("message digest").toString()<<
endl;
             cout <<"SHA-1(\"abcdefghijklmnopqrstuvwxyz\") = "<< SHA-1("abcdefghijklmn
opqrstuvwxyz").toString()<< endl;
```

```
            cout <<"SHA - 1(\"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz012345
6789\")"<< endl;
            cout <<" = "<< SHA - 1("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0
123456789").toString()<< endl;
            cout <<"SHA - 1(\"123456789012345678901234567890123456789012345678901234567890123456
78901234567890123456789 0\")"<< endl;
cout <<" = "<< SHA - 1("1234567890123456789012345678901234567890123456789012345678901234
5678901234567890").toString()<< endl;
}
//计算指定文件的 SHA - 1 摘要,并显示出来
        if((argc == 3)&&(!strcmp(pCompute,argv[1])))
        {
            //如果没有文件路径,则参数出错
            if(argv[2] == NULL)
            {
                cout <<"Parameter Error ! Please input file path !"<< endl;
                return - 1;
            }
            else
            {
                pFilePath = argv[2];
            }
            //打开指定的文件
            ifstream File_1(pFilePath);
            //声明 SHA - 1 对象,并进行计算
            SHA - 1 SHA - 1_obj1(File_1);
            //输出计算结果
            cout <<"SHA - 1(\""<< argv[2]<<"\") = "<< SHA - 1_obj1.toString()<< endl;
        }
            //手动进行文件完整性检测
        if((argc == 3)&&(!strcmp(pMValidate,argv[1])))
        {
            //如果没有文件路径,则参数出错
            if(argv[2] == NULL)
            {
                cout <<"Parameter Error ! Please input file path !"<< endl;
                return - 1;
            }
            else
            {
                pFilePath = argv[2];
            }
        //手动输入了被测文件的 SHA - 1 摘要
            cout <<"Please input the SHA - 1 value of file(\""<< pFilePath <<"\")..."<< endl;
            cin >> SHA - 1Digest;
            //在摘要的字符串末尾加上结束符
            SHA - 1Digest[32] = '\0';
            //打开指定的文件
            ifstream File_2(pFilePath);
            //声明 SHA - 1 对象,并进行计算
```

```
            //SHA-1 SHA-1_obj2(File_2);
            SHA-1 SHA-1_obj2;
            SHA-1_obj2.reset();
            SHA-1_obj2.update(File_2);
            //读取文件内容并计算 SHA-1 摘要
strTmp = SHA-1_obj2.toString( );
const char * pSHA-1Digest = strTmp.c_str( );
//输出两个摘要
cout <<"The SHA-1 digest of file(\""<< pFilePath <<"\") which you input is: "<< endl;
cout << SHA-1Digest << endl;
cout <<"The SHA-1 digest of file(\""<< pFilePath <<"\") which calculate by program is: "<< endl;
            cout << strTmp << endl;
                        //比较摘要的结果是否相同
            if (strcmp(pSHA-1Digest,SHA-1Digest))
            {
                  cout <<"Match Error! The file is not integrated!"<< endl;
            }
            else
            {
                  cout <<"Match Successfully! The file is integrated!"<< endl;
            }
        }
        //通过 SHA-1 文件进行文件完整性检测
        if((argc == 4)&&(!strcmp(pfValidate,argv[1])))
        {
            //如果没有文件路径,则参数出错
            if((argv[2] == NULL) || (argv[3] == NULL))
            {
                  cout <<"Parameter Error ! Please input file path !"<< endl;
                  return -1;
            }
            else
            {
                  pFilePath = argv[2];
                  pSHA-1FilePath = argv[3];
            }
        //打开 SHA-1 文件
            ifstream File_3(pSHA-1FilePath);
            //读取 SHA-1 文件中的记录
            File_3.getline(SHA-1Record,50);
        //以空格为标记,获得 SHA-1 文件中的 SHA-1 值与对应文件名
        char * pSHA-1Digest_f = strtok(SHA-1Record,pSpace);
        char * pFileName_f = strtok(NULL,pSpace);
                //打开被测文件
          ifstream File_4(pFilePath);
            //声明 SHA-1 对象,并进行计算
            //SHA-1 SHA-1_obj3(File_4);
            SHA-1 SHA-1_obj3;
            SHA-1_obj3.reset();
            SHA-1_obj3.update(File_4);
```

```
            //读取文件内容并计算 SHA - 1 摘要
                strTmp = SHA - 1_obj3.toString();
                const char * pSHA - 1Digest_c = strTmp.c_str();
                //输出两个摘要
                cout <<"The SHA - 1 digest of file(\""<< pFileName_f <<"\") which is in file(\""<<
pSHA - 1FilePath <<"\") is: "<< endl;
                cout<< pSHA - 1Digest_f << endl;
                cout<<" The SHA - 1 digest of file(\""<< pFilePath <<"\") which calculate by
programme is: "<< endl;
                cout<< strTmp << endl;
                    //比较摘要,进行验证
    if (strcmp(pSHA - 1Digest_c, pSHA - 1Digest_f))
        {
                cout <<"Match Error! The file is not integrated! "<< endl;
        }
        else
        {
                cout <<"Match Successfully! The file is integrated! "<< endl;
        }
    }
    //函数返回
    return 0;
}
```

通过 SHA-1 文件进行验证，与手工输入验证类似，也可以分为以下 6 个步骤。

(1) 首先比较参数 argv[1]，判断是否通过 SHA-1 文件进行验证。若是，则继续下面的步骤；否则，退出。

(2) 检测被测文件的路径和 SHA-1 文件的路径是否存在，若存在，则继续下面的步骤；否则，退出。

(3) 打开 SHA-1 文件，读取文件中的记录，调用 strtok 函数获得被测文件的 SHA-1 摘要。

(4) 打开被测文件，读取被测文件的内容，并调用 Update 函数重新计算被测文件的 SHA-1 摘要。

(5) 调用 Tostring 函数将 SHA-1 摘要表示成十六进制字符串的形式。

(6) 最后调用 strcmp 函数判断两个摘要是否相同，若相同，则说明被测文件是完整的；否则，说明文件受到了破坏。

3.3 基于 RSA 算法实现数据加解密

该算法要求实现以下目的。

(1) 理解 RSA 算法的基本工作原理。

(2) 掌握实现 RSA 算法的编程方法。

(3) 掌握基于 RSA 算法实现数据加解密的工作原理和实现方法。

RSA 算法是一种非对称加密算法，它大概是世界上使用最为广泛的公钥密码体制了，当然，它也是最著名的，因为它能够同时提供数字签名方案和公钥加密方案，从而使其成为一个非常通用的算法工具。

3.3.1 RSA 算法原理

此部分主要讲述了 RSA 算法的三部分内容，分别是 RSA 算法数学理论基础，RSA 算法的原理实现以及基于 RSA 算法实现数据加解密。

1. RSA 算法的数学基础

RSA 算法具有非常严谨的数学理论基础，现描述如下。

(1) **定理 1**　算术基本定理。任何一个不等于 0 的正整数 n 都可以写成唯一的表达式，即

$$n = p_1^{k_1} \cdots p_r^{k_r} \tag{3-1}$$

这里 $p_r > p_2 > p_3 > \cdots > p_r$ 是素数，其中，$k_i > 0$。

(2) **定义 1**　欧拉函数 $\varphi(n)$。$\varphi(n)$ 是少于或等于 n 的数中与 n 互质的数的数目，将 n 分解为素数互乘的形式 $n = p_1^{k_1} \cdots p_r^{k_r}$，每个 p_i 都是素数，则

$$\varphi(n) = p_1^{k_1}\left(1-\frac{1}{p_1}\right)p_2^{k_2}\left(1-\frac{1}{p_2}\right)\cdots p_r^{k_r}\left(1-\frac{1}{p_r}\right) \tag{3-2}$$

由此可以得出以下两条结论。

① 若 n 为质数，则 $\varphi(n) = n-1$；

② 若 m 与 n 互质，则 $\varphi(mn) = \varphi(m)\varphi(n)$。

(3) **定理 2**　欧拉定理。若整数 a 与整数 n 互素，则 $a^{\varphi(n)} \equiv 1(\bmod n)$。该定理有以下三个推论。

① 当 p 为素数时，且 $n = p$ 时，有 $a^{p-1} \equiv 1(\bmod p)$，即为 Fermat 定理：

$$a^{\varphi(n+1)} \equiv a(\bmod \mathrm{n}) \tag{3-3}$$

② 若 $n = pq$，且 p 与 q 为互异素数，取 $0 < m < n$，若 $(m,n) = 1$，有 $m^{\varphi(n)} \equiv m(\bmod n)$，即为

$$m^{(p-1)(q-1)+1} \equiv m(\bmod n) \tag{3-4}$$

2. RSA 算法的原理

RSA 算法主要包括三个部分：公私密钥的生成、加密过程及解密过程，具体算法过程如下。

1) 公钥和私钥的产生

假设 Alice 想要通过一个不可靠的媒体接收 Bob 的一条私人讯息，她可以用以下的方式来产生一个公钥和一个私钥。

(1) 随意选择两个大的质数 p 和 q，p 不等于 q，计算 $N = pq$；

(2) 根据欧拉函数，不大于 N 且与 N 互质的整数个数为 $(p-1)(q-1)$；

(3) 选择一个整数 e 与 $(p-1)(q-1)$ 互质，并且 e 小于 $(p-1)(q-1)$；

(4) 用以下公式计算 d：

$$d * e \equiv 1(\bmod (p-1)(q-1)) \tag{3-5}$$

(5) 将 p 和 q 的记录销毁。

(N,e)是公钥，(N,d)是私钥。(N,d)是秘密的。Alice 将她的公钥(N,e)传给 Bob，而将她的私钥(N,d)藏起来。

2) 加密消息

假设 Bob 想给 Alice 发送一个消息 m，他知道 Alice 产生的 N 和 e。他使用起先与 Alice 约好的格式将消息 m 转换为一个小于 N 的整数 n，比如他可以将 m 中的每一个字转换为相应的 Unicode 码，然后将这些 Unicode 码连在一起组成一个数字。假如他的信息非常长的话，他可以将这个信息分为几段，然后将每一段信息字符加密。用下面这个公式他可以将 n 加密为 c：

$$n^e \equiv c(\text{mod } N) \tag{3-6}$$

计算 c 并不复杂，Bob 算出 c 后就可以将它传递给 Alice。

3) 解密消息

Alice 得到 Bob 的消息 c 后就可以利用她的密钥 d 来解码。她可以用以下这个公式来将 c 转换为 n。

$$c^d \equiv n(\text{mod } N) \tag{3-7}$$

得到 n 后，可以将原来的信息 m 重新复原。

解码的原理是：$c^d \equiv n^{e \cdot d}(\text{mod } N)$和 $ed \equiv 1(\text{mod}(q-1))$以及 $ed \equiv 1(\text{mod}(p-1))$。由费马小定理可证明(因为 p 和 q 是质数)：

$$n^{e \cdot d} \equiv n(\text{mod } p) \quad 和 \quad n^{e \cdot d} \equiv n(\text{mod } q) \quad 以及 \quad n^{e \cdot d} \equiv n(\text{mod } pq)$$

3. RSA 算法的安全性

理论上，RSA 的安全性取决于因式分解模 n 的困难性。从严格的技术角度上来说这是不正确的，在数学上至今还未证明分解模数就是攻击 RSA 的最佳方法。事实情况是，大整数因子分解问题过去数百年来一直是令数学家头疼而未能有效解决的世界性难题。人们设想了一些非因子分解的途径攻击 RSA 体制，但这些方法都不比分解 n 来得容易。因此，严格地说，RSA 的安全性基于求解其单向函数的逆的困难性。RSA 单向函数求逆的安全性没有真正因式分解模数 n 的安全性高，而且目前人们也无法证明这两者等价。许多研究人员都试图改进 RSA 体制使它的安全性等价于因式分解模数 n。

4. 针对 RSA 算法的攻击手段

下面列出了常见的针对 RSA 算法的攻击方法。

1) 对 RSA 分解模数 n 攻击

分解模数 n 是最直接的攻击方法，也是最困难的方法。攻击者可以获得公开密钥 e 和模数 n；如果模数 $n=pq$ 被因式分解，则攻击者通过 p、q 便可计算出 $f(n)=(p-1)(q-1)$，进而 $de \equiv 1(\text{mod } f(n))$而得到解密密钥 d。如果密码分析者能够不分解 n 而直接求得 $f(n)$，则可根据 $de=1(\text{mod } f(n))$求得解密密钥 d，从而破译 RSA。又因为：

$$p+q=n-f(n)+1$$

$$p-q=\sqrt{(p+q)^2-4n}$$

所以知道 $f(n)$和 n 就可以容易地求得 p 和 q，从而成功地分解 n。目前已经出现了不对 n 进行因子分解而直接估算 $f(n)$的攻击方法，但还没证明，直接计算 $f(n)$比对 n 进行因子分

解更容易。大整数分解研究一直是数论与密码理论研究的重要课题,随着计算能力的增强和因子分解算法的不断完善,为保证RSA的安全性,在实际应用中对 p 和 q 的选取要求也越来越高。

2) RSA的小指数攻击

这类攻击专门针对RSA算法的实现细节。采用小的 e、d 可以加快加密和验证签名的速度,而且所需的存储空间小,但是如果 e、d 太小,则容易受到小指数攻击(Low Encryption Decryption Exponent Attack),包括低加密指数攻击和低解密指数攻击。通过独立随机数字对明文消息 x 进行填充,可以有效地抗击小指数攻击。

3) 耗时攻击

这种攻击是通过监视一台计算机解密消息所花费的时间来确定解密指数。RSA的基本运算是乘方取模,这种运算的特点是耗费时间精确,这样如果破译者能够监视到RSA解密的过程,并对它计时,他就能算出 d。关于如何防御这种攻击,最简单的方法莫过于使RSA解密时花费均等的时间,而与解密指数 d 无关。其次在加密前对数据做一个变换(花费恒定时间),在解密时做逆变换,这样总时间也不再依赖于解密指数 d。另外,耗时攻击对攻击者资源的要求太高,目前还不实用,但从理论上说是一个崭新的思路。

3.3.2 基于RSA算法实现数据加解密

根据上述RSA算法原理、RSA算法实现过程以及基于RSA算法实现数据加解密的实现代码如下。

```
//RSA.CPP:定义控制面板应用程序的入口点
#include "stdafx.h"
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<time.h>
#include<math.h>
#include<malloc.h>
#define MAX 100
#define LEN sizeof(struct slink)
void sub(int a[MAX], int b[MAX], int c[MAX]);
struct slink
{
int bignum[MAX];
//bignum[98]用来标记正负号,1正,0负 bignum[99]来标记实际长
struct slink *next;
};
//大数运算
void print(int a[MAX])
{
int i;
for (i = 0; i<a[99]; i++)
      printf("%d", a[a[99] - i - 1]);
printf("\n\n");
return;
```

```
}
int cmp(int a1[MAX], int a2[MAX])
{
int l1, l2;
int i;
l1 = a1[99];
l2 = a2[99];
if (l1 > l2)
      return 1;
if (l1 < l2)
      return -1;
for (i = (l1 - 1); i >= 0; i--)
{
      if (a1[i]> a2[i])
            return 1;
      if (a1[i]< a2[i])
            return - 1;
}
return 0;
}
void mov(int a[MAX], int * b)
{
int j;
for (j = 0; j < MAX; j++)
      b[j] = a[j];
return;
}
//大数相乘(向左移)
void mul(int a1[MAX], int a2[MAX], int * c)
{
int i, j, y, x, z, w, l1, l2;
l1 = a1[MAX - 1];
l2 = a2[MAX - 1];
if (a1[MAX - 2] == '-'&& a2[MAX - 2] == '-')
      c[MAX - 2] = 0;
else if (a1[MAX - 2] == '-')
      c[MAX - 2] = '-';
else if (a2[MAX - 2] == '-')
      c[MAX - 2] = '-';
for (i = 0; i < l1; i++)
{
    for (j = 0; j < l2; j++)
    {
          x = a1[i] * a2[j];
          y = x / 10;
          z = x % 10;
          w = i + j;
          c[w] = c[w] + z;
          c[w + 1] = c[w + 1] + y + c[w] / 10;
          c[w] = c[w] % 10;
```

```
        }
    }
    w = l1 + l2;
    if (c[w - 1] == 0)w = w - 1;
    c[MAX - 1] = w;
    return;
    }
    //大数相加,注意进位
    void add(int a1[MAX], int a2[MAX], int * c)
    {
    int i, l1, l2;
    int len, temp[MAX];
    int k = 0;
    l1 = a1[MAX - 1];
    l2 = a2[MAX - 1];
    if ((a1[MAX - 2] == '-') && (a2[MAX - 2] == '-'))
    {
          c[MAX - 2] = '-';
    }
    else if (a1[MAX - 2] == '-')
    {
        mov(a1, temp);
        temp[MAX - 2] = 0;
        sub(a2, temp, c);
        return;
    }
    else if (a2[MAX - 2] == '-')
    {
        mov(a2, temp);
        temp[98] = 0;
        sub(a1, temp, c);
        return;
    }
    if (l1 < l2)len = l1;
    else len = l2;
    for (i = 0; i < len; i++)
    {
        c[i] = (a1[i] + a2[i] + k) % 10;
        k = (a1[i] + a2[i] + k) / 10;
    }
    if (l1 > len)
    {
        for (i = len; i < l1; i++)
        {
              c[i] = (a1[i] + k) % 10;
              k = (a1[i] + k) / 10;
        }
        if (k != 0)
        {
            c[l1] = k;
```

```
            len = l1 + 1;
        }
        else len = l1;
    }
    else
    {
        for (i = len; i < l2; i++)
        {
              c[i] = (a2[i] + k) % 10;
              k = (a2[i] + k) / 10;
        }
        if (k != 0)
        {
            c[l2] = k;
            len = l2 + 1;
        }
        else len = l2;
    }
    c[99] = len;
    return;
    }
    //大数相减,注意借位
    void sub(int a1[MAX], int a2[MAX], int * c)
    {
    int i, l1, l2;
    int len, t1[MAX], t2[MAX];
    int k = 0;
    l1 = a1[MAX - 1];
    l2 = a2[MAX - 1];
    if ((a1[MAX - 2] == '-') && (a2[MAX - 2] == '-'))
    {
        mov(a1, t1);
        mov(a2, t2);
        t1[MAX - 2] = 0;
        t2[MAX - 2] = 0;
        sub(t2, t1, c);
        return;
    }
    else if (a2[MAX - 2] == '-')
    {
        mov(a2, t2);
        t2[MAX - 2] = 0;
        add(a1, t2, c);
        return;
    }
    else if (a1[MAX - 2] == '-')
    {
        mov(a2, t2);
        t2[MAX - 2] = '-';
        add(a1, t2, c);
```

```
        return;
    }
    if (cmp(a1, a2) == 1)
    {
        len = l2;
        for (i = 0; i < len; i++)
        {
            if ((a1[i] - k - a2[i]) < 0)
            {
                c[i] = (a1[i] - a2[i] - k + 10) % 10;
                k = 1;
            }
            else
            {
                c[i] = (a1[i] - a2[i] - k) % 10;
                k = 0;
            }
        }
        for (i = len; i < l1; i++)
        {
            if ((a1[i] - k) < 0)
            {
                c[i] = (a1[i] - k + 10) % 10;
                k = 1;
            }
            else
            {
                c[i] = (a1[i] - k) % 10;
                k = 0;
            }
        }
    if (c[l1 - 1] == 0)     //使得数组C中的前面所有0字符不显示了,如1000-20=0980-->显
                            //示为980了
        {
            len = l1 - 1;
            i = 2;
            while (c[l1 - i] == 0)//111456-111450=00006,消除0后变成了6
            {
                len = l1 - i;
                i++;
            }
        }
        else
        {
            len = l1;
        }
    }
    else
        if (cmp(a1, a2) == (-1))
        {
```

```
        c[MAX - 2] = '-';
        len = l1;
        for (i = 0; i<len; i++)
        {
            if ((a2[i] - k - a1[i])<0)
            {
                  c[i] = (a2[i] - a1[i] - k + 10) % 10;
                  k = 1;
            }
            else
            {
                  c[i] = (a2[i] - a1[i] - k) % 10;
                  k = 0;
            }
        }
        for (i = len; i<l2; i++)
        {
            if ((a2[i] - k)<0)
            {
                  c[i] = (a2[i] - k + 10) % 10;
                  k = 1;
            }
            else
            {
                  c[i] = (a2[i] - k) % 10;
                  k = 0;
            }
        }
        if (c[l2 - 1] == 0)
        {
            len = l2 - 1;
            i = 2;
            while (c[l1 - i] == 0)
            {
                len = l1 - i;
                i++;
            }
        }
        else len = l2;
    }
    else if (cmp(a1, a2) == 0)
    {
        len = 1;
        c[len - 1] = 0;
    }
c[MAX - 1] = len;
return;
}
//取模数
```

```
void mod(int a[MAX], int b[MAX], int * c)    //c = a mod b,注意：根据检验知道此处 A 和 C 的
                                             //数组都改变了
{
int d[MAX];
mov(a, d);
while (cmp(d, b) != (-1))                    //c = a - b - b - b - b - b … until(c < b)
{
    sub(d, b, c);
    mov(c, d);                               //c 复制给 a
}
return;
}
//大数相除(向右移)
//试商法,调用以后 w 为 a mod b, C 为 a div b
void divt(int t[MAX], int b[MAX], int * c, int * w)
{
int a1, b1, i, j, m;                         //w 用于暂时保存数据
int d[MAX], e[MAX], f[MAX], g[MAX], a[MAX];
mov(t, a);
for (i = 0; i < MAX; i++)
    e[i] = 0;
for (i = 0; i < MAX; i++)
    d[i] = 0;
for (i = 0; i < MAX; i++) g[i] = 0;
a1 = a[MAX - 1];
b1 = b[MAX - 1];
if (cmp(a, b) == (-1))
{
    c[0] = 0;
    c[MAX - 1] = 1;
    mov(t, w);
    return;
}
else if (cmp(a, b) == 0)
{
    c[0] = 1;
    c[MAX - 1] = 1;
    w[0] = 0;
    w[MAX - 1] = 1;
    return;
}
m = (a1 - b1);
for (i = m; i >= 0; i--)//例如//341245/3 = 341245 - 300000 * 1 ---> 41245 - 30000 * 1 --->
11245 - 3000 * 3 ---> 2245 - 300 * 7 ---> 145 - 30 * 4 = 25 --->//25 - 3 * 8 = 1
{
    for (j = 0; j < MAX; j++)
        d[j] = 0;
    d[i] = 1;
    d[MAX - 1] = i + 1;
    mov(b, g);
```

```
    mul(g, d, e);
    while (cmp(a, e) != (-1))
    {
        c[i]++;
        sub(a, e, f);
        mov(f, a);                              //f 复制给 g
    }
    for (j = i; j<MAX; j++)                     //高位清零
        e[j] = 0;
}
mov(a, w);
if (c[m] == 0) c[MAX - 1] = m;
else c[MAX - 1] = m + 1;
return;
}
//解决了 m = a * b mod n
void mulmod(int a[MAX], int b[MAX], int n[MAX], int *m)
{
int c[MAX], d[MAX];
int i;
for (i = 0; i<MAX; i++)
    d[i] = c[i] = 0;
mul(a, b, c);
divt(c, n, d, m);
for (i = 0; i<m[MAX - 1]; i++)
    printf("%d", m[m[MAX - 1] - i - 1]);
printf("\nm length is : %d \n", m[MAX - 1]);
}
//解决了 m = a ^p mod n 的函数问题
void expmod(int a[MAX], int p[MAX], int n[MAX], int *m)
{
int t[MAX], l[MAX], temp[MAX];                  //t 放入 2,l 放入 1
int w[MAX], s[MAX], c[MAX], b[MAX], i;
for (i = 0; i<MAX - 1; i++)
    b[i] = l[i] = t[i] = w[i] = 0;
t[0] = 2; t[MAX - 1] = 1;
l[0] = 1; l[MAX - 1] = 1;
mov(l, temp);
mov(a, m);
mov(p, b);
while (cmp(b, l) != 0)
{
    for (i = 0; i<MAX; i++)
        w[i] = c[i] = 0;
    divt(b, t, w, c);                           //c = p mod 2 w = p /2
    mov(w, b);                                  //p = p/2
    if (cmp(c, l) == 0)                         //余数 c == 1
    {
        for (i = 0; i<MAX; i++)
            w[i] = 0;
```

```
            mul(temp, m, w);
            mov(w, temp);
            for (i = 0; i < MAX; i++)
                w[i] = c[i] = 0;
            divt(temp, n, w, c);        //c 为余 c = temp % n,w 为商 w = temp/n
            mov(c, temp);
        }
        for (i = 0; i < MAX; i++)
            s[i] = 0;
        mul(m, m, s);                   //s = a * a
        for (i = 0; i < MAX; i++)
            c[i] = 0;
        divt(s, n, w, c);               //w = s/n;c = s mod n
        mov(c, m);
    }
    for (i = 0; i < MAX; i++)
        s[i] = 0;
    mul(m, temp, s);
    for (i = 0; i < MAX; i++)
        c[i] = 0;
    divt(s, n, w, c);
    mov(c, m);                          //余数 s 给 m
    m[MAX - 2] = a[MAX - 2];            //为后面的汉字显示需要,用第 99 位作为标记
    return;
    //k = temp * k % n
    }
    int   is_prime_san(int p[MAX])
    {
    int i, a[MAX], t[MAX], s[MAX], o[MAX],
    for (i = 0; i < MAX; i++)
        s[i] = o[i] = a[i] = t[i] = 0,
    t[0] = 1;
    t[MAX - 1] = 1;
    a[0] = 2;                           //{ 2,3,5,7 }
    a[MAX - 1] = 1;
    sub(p, t, s);
    expmod(a, s, p, o);
    if (cmp(o, t) != 0)
    {
        return 0;
    }
    a[0] = 3;
    for (i = 0; i < MAX; i++) o[i] = 0;
    expmod(a, s, p, o);
    if (cmp(o, t) != 0)
    {
        return 0;
    }
    a[0] = 5;
    for (i = 0; i < MAX; i++) o[i] = 0;
    expmod(a, s, p, o);
```

```
if (cmp(o, t) != 0)
{
    return 0;
}
a[0] = 7;
for (i = 0; i<MAX; i++) o[i] = 0;
expmod(a, s, p, o);
if (cmp(o, t) != 0)
{
    return 0;
}
return 1;
}
//检测两个大数之间是否互质
int coprime(int e[MAX], int s[MAX])
{
int a[MAX], b[MAX], c[MAX], d[MAX], o[MAX], l[MAX];
int i;
for (i = 0; i<MAX; i++)
    l[i] = o[i] = c[i] = d[i] = 0;
o[0] = 0; o[MAX - 1] = 1;
l[0] = 1; l[MAX - 1] = 1;
mov(e, b);
mov(s, a);
do
{
    if (cmp(b, l) == 0)
    {
        return 1;
    }
    for (i = 0; i<MAX; i++)
        c[i] = 0;
    divt(a, b, d, c);
    mov(b, a);                  //b-- ->a
    mov(c, b);                  //c-- ->b
} while (cmp(c, o) != 0);
//printf("They are not coprime!\n")
return 0;
}
//产生随机素数 p 和 q
void prime_random(int *p, int *q)
{
int i, k;
time_t t;
p[0] = 1;
q[0] = 3;
//p[19]=1;
//q[18]=2;
p[MAX - 1] = 10;
q[MAX - 1] = 11;
```

```
do
{
    t = time(NULL);
    srand((unsigned long)t);
    for (i = 1; i<p[MAX - 1] - 1; i++)
    {
        k = rand() % 10;
        p[i] = k;
    }
    k = rand() % 10;
    while (k == 0)
    {
        k = rand() % 10;
    }
    p[p[MAX - 1] - 1] = k;
} while ((is_prime_san(p)) != 1);
printf("素数 p 为 : ");
for (i = 0; i<p[MAX - 1]; i++)
{
    printf(" %d", p[p[MAX - 1] - i - 1]);
}
printf("\n\n");
do
{
    t = time(NULL);
    srand((unsigned long)t);
    for (i = 1; i<q[MAX - 1]; i++)
    {
        k = rand() % 10;
        q[i] = k;
    }
} while ((is_prime_san(q)) != 1);
printf("素数 q 为 : ");
for (i = 0; i<q[MAX - 1]; i++)
{
    printf(" %d", q[q[MAX - 1] - i - 1]);
}
printf("\n\n");
return;
}
//产生与(p-1)*(q-1)互素的随机数
void erand(int e[MAX], int m[MAX])
{
int i, k;
time_t t;
e[MAX - 1] = 5;
printf("随机产生一个与(p-1)*(q-1)互素的 e :");
do
{
    t = time(NULL);
```

```
        srand((unsigned long)t);
        for (i = 0; i<e[MAX - 1] - 1; i++)
        {
            k = rand() % 10;
            e[i] = k;
        }
        while ((k = rand() % 10) == 0)
            k = rand() % 10;
        e[e[MAX - 1] - 1] = k;
    } while (coprime(e, m) != 1);
    for (i = 0; i<e[MAX - 1]; i++)
    {
        printf("%d", e[e[MAX - 1] - i - 1]);
    }
    printf("\n\n");
    return;
    }
    //根据上面的p、q和e计算密钥d
    void rsad(int e[MAX], int g[MAX], int *d)
    {
    int  r[MAX], n1[MAX], n2[MAX], k[MAX], w[MAX];
    int    i, t[MAX], b1[MAX], b2[MAX], temp[MAX];
    mov(g, n1);
    mov(e, n2);
    for (i = 0; i<MAX; i++)
        k[i] = w[i] = r[i] = temp[i] = b1[i] = b2[i] = t[i] = 0;
    b1[MAX - 1] = 0; b1[0] = 0;                 //b1 = 0;
    b2[MAX - 1] = 1; b2[0] = 1;                 //b2 = 1;
    while (1)
    {
        for (i = 0; i<MAX; i++)
            k[i] = w[i] = 0;
        divt(n1, n2, k, w);                     //k = n1/n2;
        for (i = 0; i<MAX; i++)
            temp[i] = 0;
        mul(k, n2, temp);                       //temp = k * n2;
        for (i = 0; i<MAX; i++)
            r[i] = 0;
        sub(n1, temp, r);
        if ((r[MAX - 1] == 1) && (r[0] == 0))   //r = 0
        {
            break;
        }
        else
        {
            mov(n2, n1);                        //n1 = n2;
            mov(r, n2);                         //n2 = r;
            mov(b2, t);                         //t = b2;
            for (i = 0; i<MAX; i++)
                temp[i] = 0;
```

```
            mul(k, b2, temp);               //b2 = b1 - k * b2;
            for (i = 0; i<MAX; i++)
                b2[i] = 0;
            sub(b1, temp, b2);
            mov(t, b1);
        }
    }
    for (i = 0; i<MAX; i++)
        t[i] = 0;
    add(b2, g, t);
    for (i = 0; i<MAX; i++)
        temp[i] = d[i] = 0;
    divt(t, g, temp, d);
    printf("由以上的(p-1)*(q-1)和 e 计算得出的 d : ");
    for (i = 0; i<d[MAX - 1]; i++)
        printf("%d", d[d[MAX - 1] - i - 1]);
    printf("\n\n");
    }
    //求解密密钥 d 的函数(根据欧几里得算法)
    unsigned long rsa(unsigned long p, unsigned long q, unsigned long e)    //求解密密钥 d 的函数
                                                                            //(根据欧几里得算法)
    {
    unsigned long g, k, r, n1, n2, t;
    unsigned long b1 = 0, b2 = 1;
    g = (p - 1) * (q - 1);
    n1 = g;
    n2 = e;
    while (1)
    {
        k = n1 / n2;
        r = n1 - k * n2;
        if (r != 0)
        {
            n1 = n2;
            n2 = r;
            t = b2;
            b2 = b1 - k * b2;
            b1 = t;
        }
        else
        {
            break;
        }
    }
    return (g + b2) % g;
    }
    //加密和解密
    void printbig(struct slink * h)
    {
    struct slink * p;
```

```
int i;
p = (struct slink * )malloc(LEN);
p = h;
if (h != NULL)
    do
    {
        for (i = 0; i < p -> bignum[MAX - 1]; i++)
            printf("%d", p -> bignum[p -> bignum[MAX - 1] - i - 1]);
        p = p -> next;
    }
while (p != NULL);
    printf("\n\n");
}
struct slink * input(void)          //输入明文并且返回头指针,没有加密的时候转化的数字
{
struct slink * head;
struct slink * p1, * p2;
int i, n, c, temp;
char ch;
n = 0;
p1 = p2 = (struct slink * )malloc(LEN);
head = NULL;
printf("\n请输入你所要加密的内容 .\n");
while ((ch = getchar()) != '\n')
{
    i = 0;
    c = ch;
    if (c < 0)
    {
        c = abs(c);                 //把负数取正并且做一个标记
        p1 -> bignum[MAX - 2] = '0';
    }
    else
    {
        p1 -> bignum[MAX - 2] = '1';
    }
    while (c / 10 != 0)
    {
        temp = c % 10;
        c = c / 10;
        p1 -> bignum[i] = temp;
        i++;
    }
    p1 -> bignum[i] = c;
    p1 -> bignum[MAX - 1] = i + 1;
    n = n + 1;
    if (n == 1)
        head = p1;
    else p2 -> next = p1;
    p2 = p1;
```

```
    p1 = (struct slink * )malloc(LEN);
}
p2 -> next = NULL;
return(head);
}
//加密模块,例如 C = P^e mod n
struct slink * jiami(int e[MAX], int n[MAX], struct slink * head)
{
struct slink * p;
struct slink * h;
struct slink * p1, * p2;
int m = 0, i;
printf("\n");
printf("加密后形成的密文内容:\n");
p1 = p2 = (struct slink * )malloc(LEN);
h = NULL;
p = head;
if (head != NULL)
    do
    {
        expmod(p -> bignum, e, n, p1 -> bignum);
        for (i = 0; i < p1 -> bignum[MAX - 1]; i++)
        {
            printf("%d", p1 -> bignum[p1 -> bignum[MAX - 1] - 1 - i]);
        }
        m = m + 1;
        if (m == 1)
            h = p1;
        else p2 -> next = p1;
        p2 = p1;
        p1 = (struct slink * )malloc(LEN),
        p = p -> next;
    } while (p != NULL);
    p2 -> next = NULL;
    p = h;
    printf("\n");
    return(h);
}
//解密模块,例如 P = C^d mod n
void jiemi(int d[MAX], int n[MAX], struct slink * h)
{
int i, j, temp;
struct slink * p, * p1;
char ch[65535];
p1 = (struct slink * )malloc(LEN);
p = h;
j = 0;
if (h != NULL)
    do
    {
```

```
        for (i = 0; i<MAX; i++)
            p1->bignum[i] = 0;
        expmod(p->bignum, d, n, p1->bignum);
        temp = p1->bignum[0] + p1->bignum[1] * 10 + p1->bignum[2] * 100;
        if ((p1->bignum[MAX - 2]) == '0')
        {
            temp = 0 - temp;
        }
        ch[j] = temp;
        j++;
        p = p->next;
    } while (p != NULL);
    printf("\n");
    printf("解密密文后所生成的明文:\n");
    for (i = 0; i<j; i++)
        printf("%c", ch[i]);
    printf("\n");
    return;
}
//选择一种操作
void menu()
{
printf("R--------产生 密钥对\n");
printf("T--------简单测试\n");
printf("Q--------退出\n");
printf("请选择一种操作:");
}
//主函数
void main()
{
int i;
char c;
int p[MAX], q[MAX], n[MAX], d[MAX], e[MAX], m[MAX], p1[MAX], q1[MAX];
struct slink *head, *h1, *h2;
for (i = 0; i<MAX; i++)
    m[i] = p[i] = q[i] = n[i] = d[i] = e[i] = 0;            //简单初始化一下
while (1)
{
    menu();
    c = getchar();
    getchar();                                              //接受回车符
    if ((c == 'R') || (c == 'r'))                           //操作r产生密钥对
    {
        for (i = 0; i<MAX; i++)
            m[i] = p[i] = q[i] = n[i] = d[i] = e[i] = 0;
        printf("\n\n随机密钥对产生如下: \n\n");
        prime_random(p, q);                                 //随机产生两个大素数
        mul(p, q, n);                                       //p*q
```

```
        printf("由 p、q 得出 n :");
        print(n);
        mov(p, p1);
        p1[0]--;                        //p-1
        mov(q, q1);
        q1[0]--;                        //q-1
        mul(p1, q1, m);                 //m=(p-1)*(q-1)
        erand(e, m);                    //产生一个与 m 互素的随机数
        rsad(e, m, d);                  //e * d = 1 mod m
        printf("密钥对产生完成,现在可以直接进行加解密!\n");
        printf("\n 按任意键回主菜单…");
        getchar();
    }
    else if ((c == 'T') || (c == 't'))   //加密解密测试
    {
        head = input();
        h1 = jiami(e, n, head);
        jiemi(d, n, h1);
        printf("\nRSA 测试工作完成!\n");
        printf("\n 按任意键回主菜单………");
        getchar();
    }
    else if ((c == 'Q') || (c == 'q'))               //结束退出
    {
        break;
    }
}
}
```

运行结果如图 3 5 所示。



图 3-5 RSA 算法及基于 RSA 实现数据加解密运行结果图

小　结

本章首先介绍了密码学的基本概念，讲述了经典密码算法 SHA-1 算法以及 RSA 算法的原理及实现，在此基础上基于经典密码算法 SHA-1 实现文件完整性验证，利用 RSA 算法实现数据加解密。

思 考 题

1. 密码系统的基本组成有哪些？
2. 公钥密码与对称密码的主要区别是什么？
3. 哈希函数具备哪些性质？
4. 编程实现基于口令身份识别中的口令散列过程(基于 SHA-1 算法)。
5. 随机数的用途是什么？如何编程产生随机数？
6. 编程实现基于 RSA 的文件加解密。

第 4 章　基于 OpenSSL 的网络安全编程

OpenSSL 是用于安全通信的最著名的开放库。它可以实现消息摘要、文件的加密和解密、数字证书、数字签名等功能。OpenSSL EVP 提供了丰富的各种密码学函数，将具体的各种对称算法、摘要算法以及签名/验证算法进行了封装。本章学习使用 EVP API 实现各种密码算法。

4.1　OpenSSL 概述

Eric A. Young 和 Tim J. Hudson 自 1995 年开始编写后来具有巨大影响的 SSLeay 软件包，这是一个没有太多限制的开放源代码的软件包。1998 年，OpenSSL 项目组接管了 SSLeay 的开发工作，并推出了 OpenSSL 的 0.9.1 版，到目前为止，最新版本为 1.0.2。OpenSSL 的算法已经非常完善，对 SSL 2.0、SSL 3.0、TLS 1.2 以及 DTLS 1.2 都支持。OpenSSL 支持 Linux、Windows、BSD、Mac、VMS 等平台，这使得 OpenSSL 具有广泛的适用性。

4.1.1　背景技术

SSL 协议可以在 Internet 上提供秘密性传输。Netscape 公司在推出第一个 Web 浏览器的同时，提出了 SSL 协议标准，其目标是保证两个应用间通信的保密性和可靠性，可在服务器端和用户端同时实现支持，已经成为 Internet 上保密通信的工业标准。

SSL 能使用户/服务器应用之间的通信不被攻击者窃听，并且始终对服务器进行认证，还可选择对用户进行认证。SSL 协议要求建立在可靠的传输层协议之上。SSL 协议的优势在于它是与应用层协议独立无关的，高层的应用层协议（例如：HTTP，FTP，Telnet 等）能透明地建立于 SSL 协议之上。SSL 协议在应用层协议通信之前就已经完成加密算法、通信密钥的协商及服务器认证工作。在此之后应用层协议所传送的数据都会被加密，从而保证通信的私密性。

4.1.2　OpenSSL 的特点

1. 数据保密性

信息加密就是把明码的输入文件用加密算法转换成加密的文件以实现数据的保密。加密的过程需要用到密钥来加密数据然后再解密。没有了密钥，就无法解开加密的数据。数据加密之后，只有密钥要用一个安全的方法传送。加密过的数据可以公开地传送。

2. 数据完整性

加密也能保证数据的一致性。例如，消息验证码（Message Authentication Code，MAC）

能够校验用户提供的加密信息，接收者可以用 MAC 来校验加密数据，保证数据在传输过程中没有被篡改过。

3. 安全验证

加密的另外一个用途是用来作为个人的标识，用户的密钥可以作为他的安全验证的标识。SSL 是利用公开密钥的加密算法 RSA 来作为用户端与服务器端在传送机密资料时的加密通信协定。

OpenSSL 包含一个命令行工具用来完成 OpenSSL 库中的所有功能，同时，OpenSSL 是一个强大的安全套接字层密码库，Apache 使用它加密 HTTPS，OpenSSH 使用它加密 SSH，但是，不应该只将其作为一个库来使用，它还是一个多用途的、跨平台的密码工具。

4.1.3 OpenSSL 的功能

OpenSSL 整个软件包大概可以分成三个主要的功能部分：SSL 协议库、应用程序以及密码算法库。OpenSSL 的目录结构自然也是围绕这三个功能部分进行规划的。

作为一个基于密码学的安全开发包，OpenSSL 提供的功能相当强大和全面，囊括主要的密码算法、常用的密钥和证书封装管理功能以及 SSL 协议，并提供了丰富的应用程序供测试或其他目的使用。

除了上述基本功能外，OpenSSL 也提供相应辅助功能。BIO 机制是 OpenSSL 提供的一种高层 IO 接口，该接口封装了几乎所有类型的 IO 接口，如内存访问、文件访问以及 Socket 等。这使得代码的重用性大幅度提高，OpenSSL 提供 API 的复杂性也降低了很多。

OpenSSL 对于随机数的生成和管理也提供了一整套的解决方法和支持 API 函数。随机数的好坏是决定一个密钥是否安全的重要前提。

其他功能还包括：如从口令生成密钥的 API，证书签发和管理中的配置文件机制等。

4.1.4 OpenSSL 支持的算法

1. OpenSSL 密钥证书管理

密钥和证书管理是 PKI 的一个重要组成部分，OpenSSL 为之提供了丰富的功能，支持多种标准。

首先，OpenSSL 实现了 ASN.1 的证书和密钥相关标准，提供了对证书、公钥、私钥、证书请求以及 CRL 等数据对象的 DER、PEM 和 BASE64 的编解码功能。OpenSSL 提供了产生各种公开密钥对和对称密钥的方法、函数和应用程序，同时提供了对公钥和私钥的 DER 编解码功能。并实现了私钥的 PKCS#12 和 PKCS#8 的编解码功能。OpenSSL 在标准中提供了对私钥的加密保护功能，使得密钥可以安全地进行存储和分发。

在此基础上，OpenSSL 实现了对证书的 X.509 标准编解码、PKCS#12 格式的编解码以及 PKCS#7 的编解码功能。并提供了一种文本数据库，支持证书的管理功能，包括证书

密钥产生、请求产生、证书签发、吊销和验证等功能。

事实上，OpenSSL 提供的 CA 应用程序就是一个小型的证书管理中心（Certification Authority，CA），实现了证书签发的整个流程和证书管理的大部分机制。

2. SSL 和 TLS 协议

OpenSSL 实现了 SSL 协议的 SSLv2 和 SSLv3，支持了其中绝大部分算法协议。OpenSSL 也实现了 TLSv1.0，TLS 是 SSLv3 的标准化版，虽然区别不大，但毕竟有很多细节不尽相同。

虽然已经有众多的软件实现了 OpenSSL 的功能，但是 OpenSSL 里面实现的 SSL 协议能够让我们对 SSL 协议有一个更加清楚的认识，因为至少存在两点：一是 OpenSSL 实现的 SSL 协议是开放源代码的，我们可以追究 SSL 协议实现的每一个细节；二是 OpenSSL 实现的 SSL 协议是纯粹的 SSL 协议，没有跟其他协议（如 HTTP）结合在一起，澄清了 SSL 协议的本来面目。

3. OpenSSL 对称加密

OpenSSL 一共提供了 8 种对称加密算法，其中 7 种是分组加密算法，仅有的一种流加密算法是 RC4。这 7 种分组加密算法分别是 AES、DES、Blowfish、CAST、IDEA、RC2、RC5，都支持电子密码本模式（ECB）、加密分组链接模式（CBC）、加密反馈模式（CFB）和输出反馈模式（OFB）4 种常用的分组密码加密模式。其中，AES 使用的加密反馈模式（CFB）和输出反馈模式（OFB）分组长度是 128 位，其他算法使用的则是 64 位。事实上，DES 算法里面不仅是常用的 DES 算法，还支持三个密钥和两个密钥的 3DES 算法。

4. OpenSSL 非对称加密

OpenSSL 一共实现了 4 种非对称加密算法，包括 DH 算法、RSA 算法、DSA 算法和椭圆曲线算法（EC）。DH 算法一般用于密钥交换。RSA 算法既可以用于密钥交换，也可以用于数字签名，当然，如果能够忍受其缓慢的速度，那么也可以用于数据加密。DSA 算法则一般只用于数字签名。

5. 信息摘要

OpenSSL 实现了 5 种信息摘要算法，分别是 MD2、MD5、MDC2、SHA（SHA-1）和 RIPEMD。SHA 算法事实上包括 SHA 和 SHA-1 两种信息摘要算法。此外，OpenSSL 还实现了 DSS 标准中规定的两种信息摘要算法 DSS 和 DSS1。

4.1.5　OpenSSL 应用程序

OpenSSL 的应用程序已经成为 OpenSSL 一个重要的组成部分，其重要性恐怕是 OpenSSL 的开发者开始没有想到的。如 OpenCA，就是完全使用 OpenSSL 的应用程序实现的。OpenSSL 的应用程序是基于 OpenSSL 的密码算法库和 SSL 协议库写成的，所以也是一些非常好的 OpenSSL 的 API 使用范例，读懂所有这些范例，对 OpenSSL 的 API 使用就了解得比较全面了。

OpenSSL 的应用程序提供了相对全面的功能，在相当多的人看来，OpenSSL 已经为自

己做好了一切，不需要再做更多的开发工作了，所以，他们也把这些应用程序称为OpenSSL的指令。OpenSSL的应用程序主要包括密钥生成、证书管理、格式转换、数据加密和签名、SSL测试以及其他辅助配置功能。

4.1.6　OpenSSL的Engine机制

Engine机制的出现是在OpenSSL的0.9.6版的事情，开始的时候是将普通版本跟支持Engine的版本分开的，到了OpenSSL的0.9.7版，Engine机制集成到了OpenSSL的内核中，成为OpenSSL不可缺少的一部分。Engine机制的目的是为了使OpenSSL能够透明地使用第三方提供的软件加密库或者硬件加密设备进行加密。OpenSSL的Engine机制成功地达到了这个目的，这使得OpenSSL已经不仅仅是一个加密库，而是提供了一个通用的加密接口，能够与绝大部分加密库或者加密设备协调工作。当然，要使特定加密库或加密设备OpenSSL协调工作，需要编写少量的接口代码，但是这样的工作量并不大，虽然还是需要一点儿密码学的知识。Engine机制的功能跟Windows提供的CSP功能目标基本是相同的，包括CryptoSwift、nCipher、Atalla、Nuron、UBSEC、Aep、SureWare以及IBM 4758 CCA的硬件加密设备。当然，所有上述Engine接口支持不一定很全面，比如，可能支持其中一两种公开密钥算法。

4.1.7　OpenSSL安装方法

对应不同的操作系统，用户可以参考INSTALL、INSTALL.MacOS、INSTALL.NW、INSTALL.OS2、INSTALL.VMS、INSTALL.W32、INSTALL.W64和INSTALL.WCE等文件来安装OpenSSL。安装时，需要如下条件：Make工具、Perl 5、编译器以及C语言库和头文件。

1. Linux下的安装

(1) 解压OpenSSL开发包文件。

(2) 运行./config --prefix=/usr/local/openssl (更多选项用./config --help来查看)，可用的选项有：no-mdc2、no-cast no-rc2、no-rc5、no-ripemd、no-rc4 no-des、no-md2、no-md4、no-idea、no-aes、no-bf、no-err、no-dsa、no-dh、no-ec、no-hw、no-asm、no-krb5、no-dso、no-threads、no-zlib、DOPENSSL_NO_HASH_COMP、DOPENSSL_NO_ERR、DOPENSSL_NO_HW、DOPENSSL_NO_OCSP、DOPENSSL_NO_SHA256和DOPENSSL_NO_SHA512等。去掉不必要的内容可以减少生成库的大小。若要生成Debug版本的库和可执行程序需加-g或者-g3(OpenSSL中有很多宏，需要调试学习最好加上-g3)。

(3) make test (可选)。

(4) make install。

完成后，OpenSSL会被安装到/usr/local/openssl目录，包括头文件目录include、可执行文件目录bin、在线帮助man、库目录lib以及配置文件目录(ssl)。

2. Windows下的编译与安装

OpenSSL的编译安装需要Perl的支持，下载最新版ActivePerl和OpenSSL源码包。

安装步骤如下。

(1) 安装 VC6.0；0.9.7i 及以上版本支持 VC++2005。

(2) 安装 Perl5。

(3) 解压 OpenSSL。

(4) 在控制台下进入 openssl 目录，执行如下指令(x64 环境)。

```
>perl Configure VC-WIN64A
>ms\do_win64a
>nmake -f ms\ntdll.mak
>cd out32dll
>..mstest
```

或者执行如下指令(x32 环境)。

```
>perl configure VC-WIN32
>ms\do_ms
>nmake -f ms\ntdll.mak
>nmake -f ms\ntdll.mak test
```

(5) 运行 perl Configure VC-WIN64A 或 VC-WIN32。

(6) ms\do_ms.bak。

(7) nmake-f ms\ntdll.mak(动态库)或者 nmake-f ms\nt.mak(静态库)。

编译 Debug 版本需在 ms\do_ms.bat 中加上 debug,，见 INSTALL.W32，具体做法如下。

编辑 do_ms.bak，修改前内容如下。

```
perl util\mkfiles.pl >MINFO
perl util\mk1mf.pl no-asm VC-WIN32 >ms\nt.mak
perl util\mk1mf.pl dll no-asm VC-WIN32 >ms\ntdll.mak
perl util\mk1mf.pl no-asm VC-CE >ms\ce.mak
perl util\mk1mf.pl dll no-asm VC-CE >ms\cedll.mak
perl util\mkdef.pl 32 libeay >ms\libeay32.def
perl util\mkdef.pl 32 ssleay >ms\ssleay32.def
```

添加 debug 后内容如下。

```
perl util\mkfiles.pl >MINFO
perl util\mk1mf.pl debug no-asm VC-WIN32 >ms\nt.mak #添加 debug
perl util\mk1mf.pl debug dll no-asm VC-WIN32 >ms\ntdll.mak #添加 debug
perl util\mk1mf.pl debug no-asm VC-CE >ms\ce.mak #添加 debug
perl util\mk1mf.pl debug dll no-asm VC-CE >ms\cedll.mak #添加 debug
perl util\mkdef.pl 32 libeay >ms\libeay32.def
perl util\mkdef.pl 32 ssleay >ms\ssleay32.def
```

安装完毕后，生成的头文件放在 inc32 目录，动 静态库和可执行文件放在 outdll 目录。

配置环境过程如下。

(1) 解压 OpenSSL 压缩包，进行编译，也可以寻找编译完成的 OpenSSL 开发包直接使用。经过以下简单几步配置就可以进行基于 OpenSSL 的编程了。如果要使用命令行建议

在 Kali(或其他 Linux 发行版本)下进行,Windows 7 及以后版本的系统中 OpenSSL 命令行操作容易出错。

(2) 添加包含目录和库目录,如图 4-1 所示。



图 4-1 添加包含目录和库目录

(3) 添加预处理命令(预防报出莫名其妙的错误,如图 4-2 所示),如图 4-3 所示。

```
_WINSOCK_DEPRECATED_NO_WARNINGS_CRT_SECURE_NO_WARNINGS
```



图 4-2 错误提示

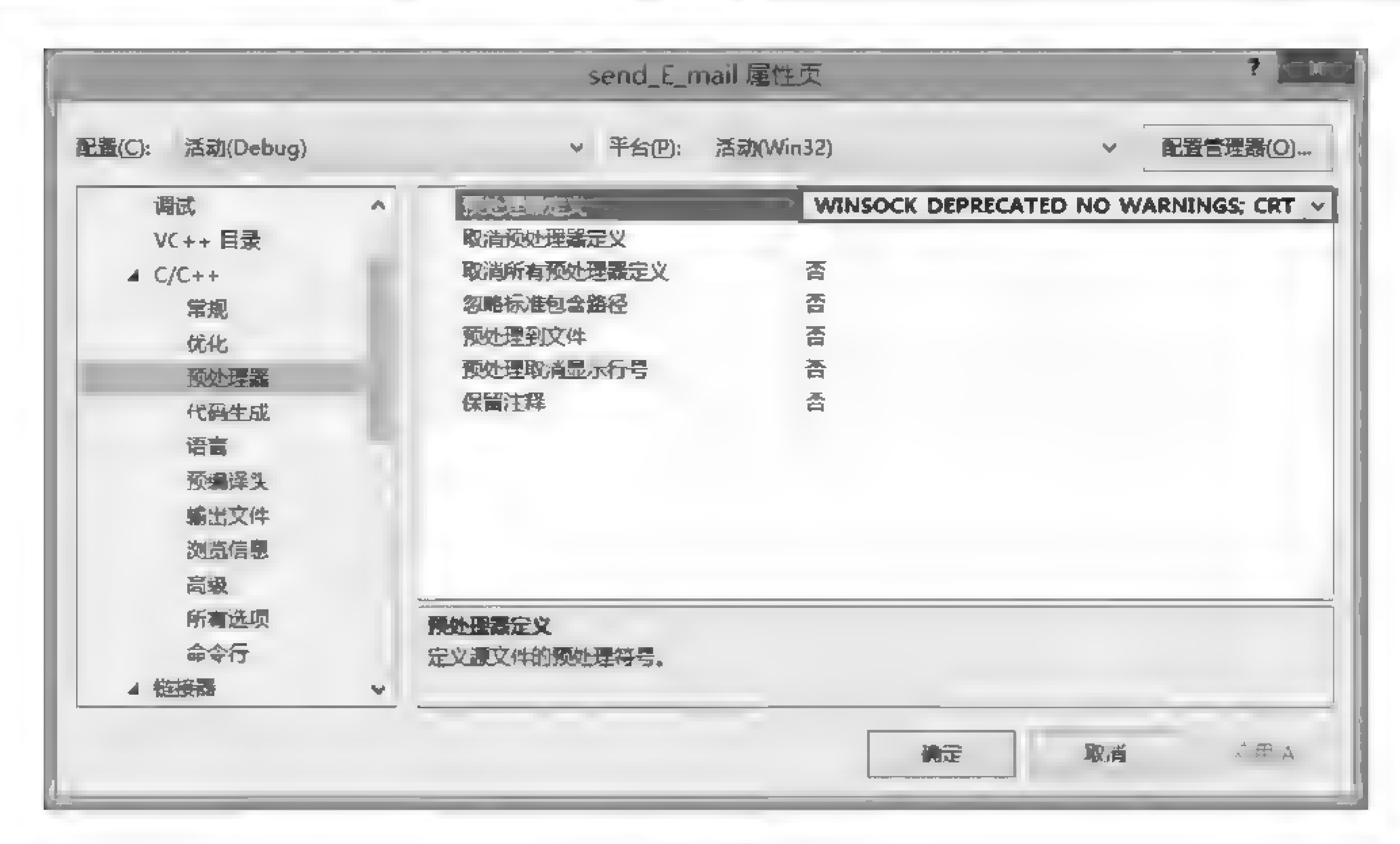


图 4-3　添加预处理命令

(4) 添加依赖库 ws2_32. lib, libeay32. lib, ssleay32. lib, 如图 4-4 所示。

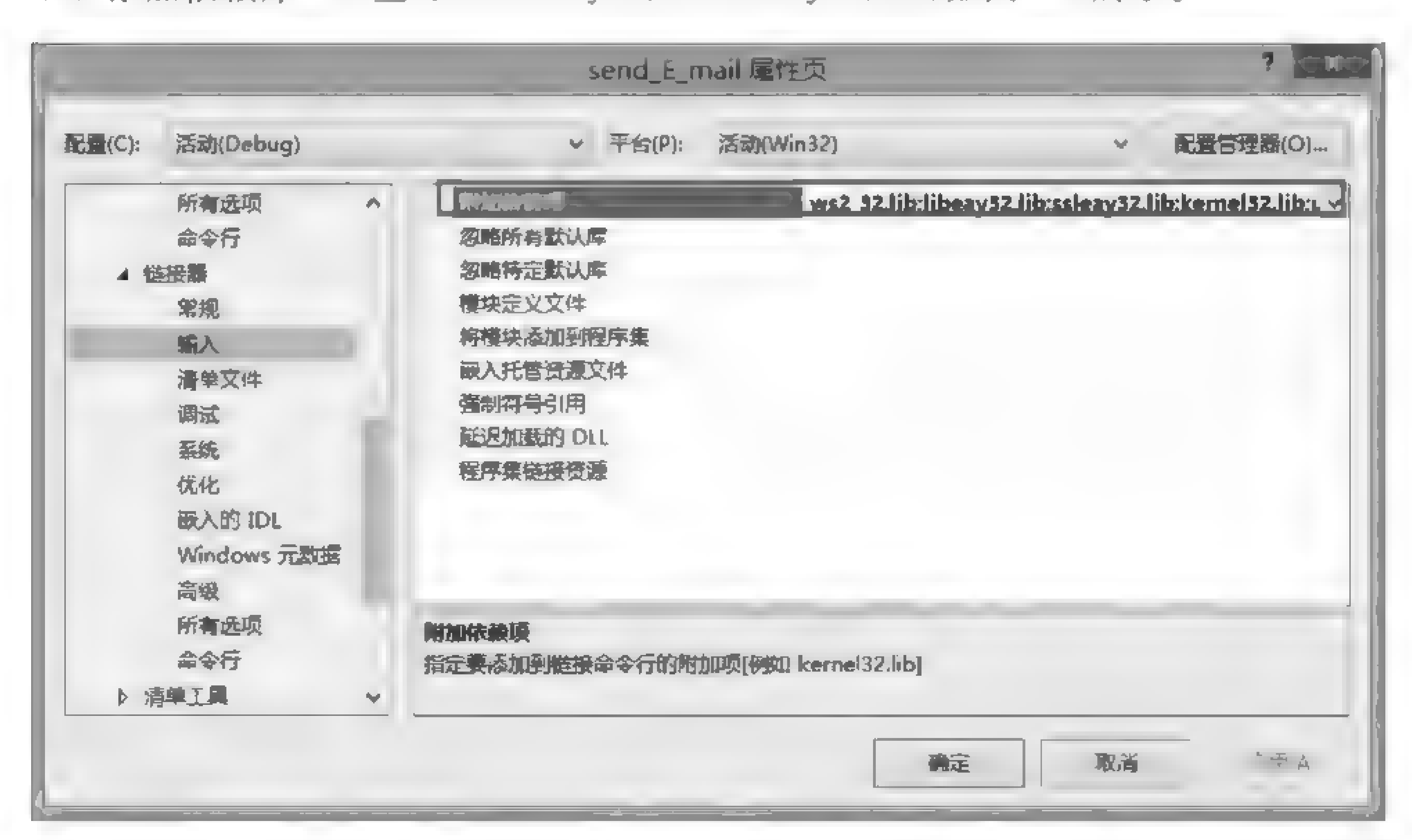


图 4-4　添加依赖库

或者直接在程序头部添加以下语句。

```
#pragma  comment(lib,"ws2_32.lib")         /*链接 ws2_32.lib 库*/
#pragma  comment(lib," libeay32.lib")      /*链接 libeay32.lib 库*/
#pragma  comment(lib," ssleay32.lib")      /*链接 ssleay32.lib 库*/
```

然后将这三个动态链接库文件(*.dll)复制到工程文件的 DEBUG 中。

具体详情可参考 openssl 目录下的 Install. w32 等文件。

4.2 OpenSSL EVP 编程

4.2.1 概述

EVP 是 OpenSSL 自定义的一组高层算法封装函数，它是对具体加密算法的高层抽象，使得可以在通信类加密算法框架下，通过相应的接口去调用不同的加密算法或者遍历地改变具体的加密算法，这样就大大提高了代码的可重用性。OpenSSL 的 EVP API 包括对称加密算法、非对称加密算法、签名验证以及信息摘要等算法的封装。

EVP 系列的函数定义包含在 crypto evp.h 里面，这是一系列封装了 OpenSSL 加密库里面所有算法的函数。通过这样的统一的封装，使得只需要在初始化参数的时候做很少的改变，就可以使用相同的代码但采用不同的加密算法进行数据的加密和解密。

OpenSSL EVP 提供了丰富的密码学中的各种函数。OpenSSL 中实现了各种对称算法、摘要算法以及签名/验证算法。EVP 函数将这些具体的算法进行了封装。EVP 主要封装了如下功能函数。

(1) 实现了 base64 编解码 BIO。

(2) 实现了加解密 BIO。

(3) 实现了摘要 BIO。

(4) 实现了 reliable BIO。

(5) 封装了摘要算法。

(6) 封装了对称加解密算法。

(7) 封装了非对称密钥的加密(公钥)、解密(私钥)、签名与验证以及辅助函数。

(8) 基于口令的加密(PBE)。

(9) 对称密钥处理。

(10) 数字信封：数字信封用对方的公钥加密对称密钥，数据则用此对称密钥加密。发送给对方时，同时发送对称密钥密文和数据密文。接收方首先用自己的私钥解密密钥密文，得到对称密钥，然后用它解密数据。

(11) 其他辅助函数。

4.2.2 源码结构

evp 源码位于 crypto/evp 目录，可以分为如下几类。

1. 全局函数

主要包括 c_allc.c、c_alld.c、c_all.c 以及 names.c。它们加载 OpenSSL 支持的所有的对称算法和摘要算法，放入到哈希表中。实现了 OpenSSL_add_all_digests、OpenSSL_add_all_ciphers 以及 OpenSSL_add_all_algorithms(调用了前两个函数)函数。在进行计算时，用户也可以单独加载摘要函数(EVP_add_digest)和对称计算函数(EVP_add_cipher)。

2. BIO 扩充

包括 bio_b64.c、bio_enc.c、bio_md.c 和 bio_ok.c，各自实现了 BIO_METHOD 方法，

分别用于 base64 编解码、对称加解密以及摘要。

3. 摘要算法 EVP 封装

由 digest.c 实现，实现过程中调用了对应摘要算法的回调函数。各个摘要算法提供了自己的 EVP_MD 静态结构，对应源码为 m_xxx.c。

4. 对称算法 EVP 封装

由 evp_enc.c 实现，实现过程调用了具体对称算法函数，实现了 Update 操作。各种对称算法都提供了一个 EVP_CIPHER 静态结构，对应源码为 e_xxx.c。需要注意的是，e_xxx.c 中不提供完整的加解密运算，它只提供基本的对于一个 block_size 数据的计算，完整的计算由 evp_enc.c 来实现。当用户想添加一个自己的对称算法时，可以参考 e_xxx.c 的实现方式。一般用户至少需要实现如下功能。

(1) 构造一个新的静态的 EVP_CIPHER 结构；

(2) 实现 EVP_CIPHER 结构中的 init 函数，该函数用于设置 iv，设置加解密标记，以及根据外送密钥生成自己的内部密钥；

(3) 实现 do_cipher 函数，该函数仅对 block_size 字节的数据进行对称运算；

(4) 实现 cleanup 函数，该函数主要用于清除内存中的密钥信息。

5. 非对称算法 EVP 封装

主要是以 p_开头的文件。其中，p_enc.c 封装了公钥加密；p_dec.c 封装了私钥解密；p_lib.c 实现一些辅助函数；p_sign.c 封装了签名函数；p_verify.c 封装了验签函数；p_seal.c 封装了数字信封；p_open.c 封装了解数字信封。

6. 基于口令的加密

包括 p5_crpt2.c、p5_crpt.c 和 evp_pbe.c。

4.2.3 对称算法以及 base64 编码编程

1. 主要数据结构和函数说明

对称加密算法封装的函数系列名字是以 EVP_Encrypt*...*开头的，其实，这些函数只是简单调用了 EVP_Cipher*...*系列的同名函数，换一个名字可能是为了更好地区别和理解。除了实现了对称加密算法外，EVP_Encrypt*...*系列还对块加密算法提供了缓冲功能。以后可能会更多使用 EVP_Cipher 的术语，因为它是真正的实现结构。EVP_Cipher 是 OpenSSL EVP 中一个非常重要的结构体。

EVP_Cipher*...*得以实现的一个基本结构是下面定义的一个算法结构，它定义了 EVP_cipher 系列函数应该采用什么算法进行数据处理，其定义如下(evp.h)。

```
typedef struct evp_cipher_st
{
    int  nid;
    int  block_size;
    int  key_len;
    int  iv_len;
    unsigned  long flags;
```

```
  int ( * init) (EVP_CIPHER_CTX * ctx,const unsigned char * key,   const unsigned char * iv,
int enc);
  int ( * do_cipher) (EVP_CIPHER_CTX * ctx,unsigned char * out,const unsigned char * in,unsi
gned int inl);
int ( * cleanup)(EVP_CIPHER_CTX * );
int ctx_size;
int ( * set_asn1_parameters)(EVP_CIPHER_CTX * ,ASN1_TYPE * );
int ( * get_asn1_parameters)(EVP_CIPHER_CTX * ,ASN1_TYPE * );
int ( * ctrl)(EVP_CIPHER_CTX * ,int type,int arg,void * ptr); /* Miscellaneous operations */
void * app_data;
} EVP_Cipher;
```

该结构用来存放对称加密相关的信息以及算法。

参数说明如下。

nid：算法类型的nid识别号，OpenSSL里面每个对象都有一个内部唯一的识别ID。

block_size：每次加密的数据块的长度，以字节为单位。

key_len：各种不同算法默认的密钥长度。

iv_len：初始化向量的长度。

init：算法结构初始化函数，可以设置为加密模式或是解密模式。

do_cipher：进行数据加密或解密的函数。

cleanup：释放EVP_CIPHER_CTX结构里面的数据和设置。

ctx_size：设定ctx->cipher_data数据的长度。

set_asn1_parameters：在EVP_CIPHER_CTX结构中通过参数设置一个ASN1_TYPE。

get_asn1_parameters：从一个ASN1_TYPE中取得参数。

ctrl：其他各种操作函数。

app_data：应用数据。

通过定义这样一个指向这个结构的指针，就可以在连接程序的时候只连接自己使用的算法；而如果是通过一个整数来指明应该使用什么算法的话，会导致所有算法的代码都被连接到代码中。通过这样一个结构，也可以增加新的算法。在此基础上，每个EVP_Encrypt都维护着一个指向EVP_CIPHER_CTX结构的指针，该结构体的定义如下。

```
typedef Struct EVP_Cipher_Ctx_st
{
const EVP_CIPHER * cipher;
ENGINE * engine;
int encrypt;
int buf_len;
unsigned char oiv[EVP_MAX_IV_LENGTH];
unsigned char iv[EVP_MAX_IV_LENGTH];
unsigned char buf[EVP_MAX_BLOCK_LENGTH];
int num;
void * app_data;
int key_len;
unsigned long flags;
void * cipher_data;
```

```
int final_used;
int block_mask;
unsigned char final[EVP_MAX_BLOCK_LENGTH];
 } EVP_CIPHER_CTX
```

对称算法上下文结构，此结构主要用来维护加解密状态，存放中间以及最后结果。因为加密或解密时，当数据很多时，可能会用到 Update 函数，并且每次加密或解密的输入数据长度是任意的，并不一定是对称算法 block_size 的整数倍，所以需要用该结构来存放中间未加密的数据。

参数说明如下。

cipher：该结构相关的一个 EVP_CIPHER 算法结构。

engine：如果加密算法是 ENGINE 提供的，那么该成员保存了相关的函数接口。

encrypt：加密或解密的标志。

buf_len：该结构缓冲区里面当前的数据长度。

oiv：初始的初始化向量。

iv：工作时候使用的初始化向量。

buf：保存下来的部分需要的数据。

num：在 cfb/ofb 模式的时候指定块长度。

app_data：应用程序要处理的数据。

key_len：密钥长度，算法不一样长度也不一样。

cipher_data：加密后的数据。

上述两个结构体是 EVP_Cipher(EVP_Encrypt)系列的两个基本结构体，其他一系列函数都是以这两个结构为基础实现的。文件 evp\evp_enc.c 是最高层的封装实现，各种加密算法封装在 p_enc.c 里面实现，解密算法封装在 p_dec.c 里面实现，而各个 e_*.c 文件则真正实现了各种算法的加解密功能，它们其实也是一些封装函数，真正的算法实现在各个算法同名目录里面的文件实现。

2. EVP_Encrypt 支持的对称加密算法

OpenSSL 对称加密算法的格式都以函数形式提供，其实该函数返回一个该算法的结构体，其形式一般如下：

```
EVP_CIPHER * EVP_* (void)
```

在 OpenSSL 中，所有提供的对称加密算法长度都是固定的，有特别说明的除外。下面对这些算法进行分类介绍，首先介绍一下算法中使用的通用标志的含义。

• 通用标志

ecb —— 电子密码本(Electronic Code Book)加密方式。

cbc——加密块链接(Cipher Block Chaining)加密方式。

cfb——64 位加密反馈(Cipher Feedback)加密方式。

ofb——64 位输出反馈(Output Feedback)加密方式。

ede —— 该加密算法采用了加密、解密、加密的方式，第一个密钥和最后一个密钥是相同的。

ede3 —— 该加密算法采用了加密、解密、加密的方式，但是三个密钥都不相同。

• NULL 算法

函数：EVP_enc_null()

说明：该算法不做任何事情，也就是没有进行加密处理。

• DES 算法

函数：EVP_des_cbc(void)，EVP_des_ecb(void)，EVP_des_cfb(void)，EVP_des_ofb(void)

说明：分别是 CBC 方式、ECB 方式、CFB 方式以及 OFB 方式的 DES 算法。

• 使用两个密钥的 3DES 算法

函数：EVP_des_ede_cbc(void)，EVP_des_ede()，EVP_des_ede_ofb(void)，EVP_des_ede_cfb(void)

说明：分别是 CBC 方式、ECB 方式、CFB 方式以及 OFB 方式的 3DES 算法，算法的第一个密钥和最后一个密钥相同，事实上就只需要两个密钥位。

• 使用三个密钥的 3DES 算法

函数：EVP_des_ede3_cbc(void)，EVP_des_ede3()，EVP_des_ede3_ofb(void)，EVP_des_ede3_cfb(void)

说明：分别是 CBC 方式、ECB 方式、CFB 方式以及 OFB 方式的 3DES 算法，算法的三个密钥都不相同。

• DESX 算法

函数：EVP_desx_cbc(void)

说明：CBC 方式 DESX 算法。

• RC4 算法

函数：EVP_rc4(void)

说明：RC4 流加密算法。该算法的密钥长度可以改变，默认是 128。

• 40 位 RC4 算法

函数：EVP_rc4_40(void)

说明：密钥长度 40 位的 RC4 流加密算法。该函数可以使用 EVP_rc4 和 EVP_CIPHER_CTX_set_key_length 函数代替。

• IDEA 算法

函数：EVP_idea_cbc()，EVP_idea_ecb(void)，EVP_idea_cfb(void)，EVP_idea_ofb(void)

说明：分别是 CBC 方式、ECB 方式、CFB 方式以及 OFB 方式的 IDEA 算法。

• RC2 算法

函数：EVP_rc2_cbc(void)，EVP_rc2_ecb(void)，EVP_rc2_cfb(void)，EVP_rc2_ofb(void)

说明：分别是 CBC 方式、ECB 方式、CFB 方式以及 OFB 方式的 RC2 算法，该算法的密钥长度是可变的，可以通过设置有效密钥长度或有效密钥位来设置参数来改变。默认的是 128 位。

• 定长的两种 RC2 算法

函数：EVP_rc2_40_cbc(void)，EVP_rc2_64_cbc(void)

说明：分别是 40 位和 64 位 CBC 模式的 RC2 算法。

• Blowfish 算法

函数：EVP_bf_cbc(void)，EVP_bf_ecb(void)，EVP_bf_cfb(void)，EVP_bf_ofb(void)

说明：分别是 CBC 方式、ECB 方式、CFB 方式以及 OFB 方式的 Blowfish 算法，该算法的密钥长度是可变的。

• CAST 算法

函数：EVP_cast5_cbc(void)，EVP_cast5_ecb(void)，EVP_cast5_cfb(void)，EVP_cast5_ofb(void)

说明：分别是 CBC 方式、ECB 方式、CFB 方式以及 OFB 方式的 CAST 算法，该算法的密钥长度是可变的。

• RC5 算法

函数：EVP_rc5_32_12_16_cbc(void)，EVP_rc5_32_12_16_ecb(void)，EVP_rc5_32_12_16_cfb(void)，EVP_rc5_32_12_16_ofb(void)

说明：分别是 CBC 方式、ECB 方式、CFB 方式以及 OFB 方式的 RC5 算法，该算法的密钥长度可以根据参数 number of rounds(算法中一个数据块被加密的次数)来设置，默认的是 128 位密钥，加密次数为 12 次。目前来说，由于 RC5 算法本身实现代码的限制，加密次数只能设置为 8、12 或 16。

• 128 位 AES 算法

函数：EVP_aes_128_ecb(void)，EVP_aes_128_cbc(void)，PEVP_aes_128_cfb(void)，EVP_aes_128_ofb(void)

说明：分别是 CBC 方式、ECB 方式、CFB 方式以及 OFB 方式的 128 位 AES 算法。

• 192 位 AES 算法

函数：EVP_aes_192_ecb(void)，EVP_aes_192_cbc(void)，PEVP_aes_192_cfb(void)，EVP_aes_192_ofb(void)

说明：分别是 CBC 方式、ECB 方式、CFB 方式以及 OFB 方式的 192 位 AES 算法。

• 256 位 AES 算法

函数：EVP_aes_256_ecb(void)，EVP_aes_256_cbc(void)，PEVP_aes_256_cfb(void)，EVP_aes_256_ofb(void)

说明：分别是 CBC 方式、ECB 方式、CFB 方式以及 OFB 方式的 256 位 AES 算法。

3. EVP_Encrypt 系列函数详解

该系列函数分为两部分，一部分是基本函数系列，另一部分是设置函数系列。基本系列函数主要是进行基本的加密和解密操作的函数，与对称算法加解密操作相关的函数主要包括以下三类：加解密初始化函数、加解密更新函数、加解密结束函数。对于每个 EVP_XXX 函数还有一个对应的 EVP_XXX_ex 版本用于支持 OpenSSL 的 ENGINE 机制，对于 EVP_XXX_ex 版本的函数，除了在 init_xx 系列函数里多一个 ENGINE * impl 参数用于指定 ENGINE 实例外，其他函数的参数列表是完全相同的。相关函数说明如下。

1) 加密函数

```
int  EVP_EncryptInit (
 EVP_CIPHER_CTX * ctx,
 const EVP_CIPHER   * type,
 const unsigned  char * key,
```

```
        const unsigned char * iv
    );
```

功能：该函数初始化一个 EVP_CIPHER_CTX 结构体，只有初始化后该结构体才能在下面介绍的函数中使用。

返回值：操作成功返回 1，否则返回 0。

参数说明如下。

ctx[in,out]：EVP-cipher 上下文对象。

type[in]：加解密所用的密码类型数据结构。

key[in]：加解密所用的密钥。

iv[in]：初始化向量，用在某些加密模式中，如 CBC。

```
int   EVP_EncryptInit_ex (
EVP_CIPHER_CTX  * ctx,
const   EVP_CIPHER   * type,
  ENGINE   * impl,
const unsigned char * key,
const unsigned char * iv
);
```

功能：该函数采用 ENGINE 参数 impl 的算法来设置并初始化加密结构体。

返回值：操作成功返回 1，否则返回 0。

参数说明如下。

ctx[in,out]：EVPcipher 上下文对象。

type：通常通过函数类型来提供参数，如 EVP_des_cbc 函数的形式，即第 3 章中介绍的对称加密算法的类型。

impl：值为 NULL，使用缺省的实现算法。

key：用来加密的对称密钥。

iv：参数是初始化向量(如果需要的话)。

在算法中真正使用的密钥长度和初始化密钥长度是根据算法来决定的。在调用该函数进行初始化的时候，除了参数 type 之外，所有其他参数可以设置为 NULL，留到以后调用其他函数的时候再提供，这时候参数 type 设置为 NULL 就可以了。在默认的加密参数不合适的时候，可以这样处理。

```
int   EVP_EncryptUpdate (
  EVP_CIPHER_CTX  * ctx,
unsigned char    * out,
int  * outl,
const unsigned char * in,
int inl
);
```

功能：该函数执行对数据的加密。

返回值：操作成功返回 1，否则返回 0。

参数说明如下。

ctx[in,out]：cipher 上下文对象。

out[out]：存放加解密结果的数据缓冲区。

outl[out]：存放加解密结果的字节长度。

in[in]：输入数据缓冲区。

inl[in]：输入数据的字节长度。

该函数加密从参数 in 输入长度为 inl 的数据，并将加密好的数据写入到参数 out 里面去。可以通过反复调用该函数来处理一个连续的数据块。写入到 out 的数据数量是由已经加密的数据的对齐关系决定的，理论上来说，从 0 到 inl + cipher_block_size-1 的任何一个数字都有可能(单位是字节)，所以输出的参数 out 要有足够的空间存储数据。写入到 out 中的实际数据长度保存在 outl 参数中。

```
int  EVP_EncryptFinal_ex (
 EVP_CIPHER_CTX * ctx,
  unsigned char   * out, int * outl
);
```

功能：结束加解密。

返回值：操作成功返回 1，否则返回 0。

参数说明如下。

ctx[in,out]：cipher 上下文对象。

out[out]：存放最后一段加密结果的数据缓冲区。

outl[out]：最后一段加密结果的字节长度。

PKCS padding 标准是这样定义的，在被加密的数据后面加上 n 个值为 n 的字节，使得加密后的数据长度为加密块长度的整数倍。无论在什么情况下，都是要加上 padding，也就是说，如果被加密的数据已经是块长度的整数倍，那么这时候 n 就应该等于块长度。比如，如果块长度是 9，要加密的数据长度是 11，那么 5 个值为 5 的字节就应该增加在数据的后面。

该函数处理最后(Final)的一段数据，在函数 padding 功能打开的时候(缺省)才有效，这时候，它将剩余的最后的所有数据进行加密处理。该算法使用标志的块 padding 方式(AKA PKCS padding)。加密后的数据写入到参数 out 里面，参数 out 的长度至少应该等于一个加密块。写入的数据长度信息输入到 outl 参数里面。该函数调用后，表示所有数据都加密完了，不应该再调用 EVP_EncryptUpdate 函数。如果没有设置 padding 功能，那么本函数不会加密任何数据，如果还有剩余的数据，那么就会返回错误信息，也就是说，这时候数据总长度不是块长度的整数倍。

2) 解密函数

```
int  EVP_Decryptinit(
  EVP_CIPHER_CTX * ctx,
  const  EVP_CIPHER * cipher,
  const unsigned char * key,
  const unsigned char * iv
  );
  int  EVP_DecryptInit_ex (
  EVP_CIPHER_CTX   * ctx,
  const  EVP_CIPHER   * cipher,
   ENGINE   * iimpl,
```

```
const unsigned char * key,
const unsigned char * iv
);
 int  EVP_DecryptUpdate (
   EVP_CIPHER_CTX * ctx,
 unsigned char  * out,
 int * outl,
 const unsigned char * in,
 int inl
 );
int  EVP_DecryptFinal _ex(
 EVP_CIPHER_CTX * ctx,
  unsigned char  * out, int * outl
);
```

这 4 个函数是上面 4 个加密函数相应的解密函数,参数说明也相同。这些函数的参数要求基本上都跟上面相应的加密函数相同。如果 padding 功能打开了,EVP_DecryptFinal 会检测最后一段数据的格式,如果格式不正确,该函数会返回错误代码。此外,如果打开了 padding 功能,EVP_DecryptUpdate 函数的参数 out 的长度应该至少为 inl+cipher_block_size 字节;但是,如果加密块的长度为 1,则其长度为 inl 字节就足够了。4 个函数都是操作成功返回 1,否则返回 0。

需要注意的是,虽然在 padding 功能开启的情况下,解密操作提供了错误检测功能,但是该功能并不能检测输入的数据或密钥是否正确,所以即便一个随机的数据块也可能无错地完成该函数的调用。如果 padding 功能关闭了,那么当解密数据长度是块长度的整数倍时,操作总是返回成功的结果。

4. 支持 base64 的主要函数

在实际的网络传输中,base64 编码应用广泛,其功能是将一个任意的字节序列编码为可打印的字符序列。base64 在 OpenSSL 中有直接的库支持,其函数命名和调用方式类似 EVP_Encrypt。

支持 base64 的主要函数由以下三类组成:编解码初始化函数,编解码更新函数,编解码结束函数。

1) 编解码初始化函数

```
int  EVP_EncodeInit(EVP_ENCODE_CTX  * ctx);
int  EVP_Decodeinit(EVP_ENCODE_CTX  * ctx );
```

功能:编解码初始化。

返回值:失败返回 0,成功返回非 0 值。

参数说明如下。

ctx[in]:EVP base64 encode 对象。

2) 编解码更新函数——用于不断加入要被编码的数据

```
int  EVP_EncodeUpdate(
  EVP_ENCODE_CTX * ctx,
  unsigned char * out,
```

```
            int  * outl,
            unsigned char   * in,
            int inl
    );
        int  EVP_DecodeUpdate(
            EVP_ENCODE_CTX   * ctx,
            unsigned char  * out,
            int  outl,
        const  unsigned char    * in,
            int  inl
            );
```

功能：编解码更新。

返回值：失败返回 0，成功返回非 0 值。

参数说明如下。

ctx[in,out]：EVP base64 encode 对象。

out[out]：存放编解码结果的数据缓冲区。

outl[out]：存放编解码结果的字节长度。

in[in]：输入数据缓冲区。

inl[in：输入数据的字节长度。

对于 EVP_DecodeUpdate 函数，需要特别说明的是其返回值，－1 表示解码出错，0 表示没有数据需要编码，1 表示还有其他行数据需要解码。

3）编解码结束函数——用于加入或者消除填充数据

```
int EVP_encodeFinal(
     EVP_ENCODE_CTX * ctx,
     unsigned char * out,
     int  outl
  );
int EVP_DecodeFinal(
    EVP_ENCODE_CTX * ctx,
    unsigned char * outm,
    int  outl
  );
```

功能：结束编解码。

返回值：失败返回 0，成功返回非 0 值。

参数说明如下。

ctx[in,out]：cipher 上下文对象。

out[out]：存放最终编码结果的数据缓冲区。

outl[out]：存放最终解码结果的字节长度。

5. 应用举例

一般来说，EVP_Encrypt * ... * 系列函数的应用架构如下所描述(假设加密算法为 3DES)。

1）定义一些必需的变量

```
char key[EVP_MAX_KEY_LENGTH];
char iv[EVP_MAX_IV_LENGTH];
EVP_CIPHER_CTX ctx;
unsigned char out[512 + 8];
int outl;
```

给变量 key 和 iv 赋值，这里使用了函数 EVP_BytesToKey，该函数从输入密码产生了密钥 key 和初始化向量 iv，该函数将在后面做介绍。如果可以有别的办法设定 key 和 iv，该函数调用则不是必需的。

```
EVP_BytesToKey(EVP_des_ede3_cbc,EVP_SHA-1,NULL,passwd,strlen(passwd),key,iv);
```

2）初始加密算法结构 EVP_CIPHER_CTX

```
EVP_EncryptInit_ex(&ctx,EVP_des_ede3_cbc(),NULL,key,iv);
```

3）进行数据的加密操作

```
while (...)
{   EVP_EncryptUpdate(ctx,out,&outl,in,512);}
```

一般来说采用循环结构进行处理，每次循环加密数据为 512 字节，密文输出到 out。out 和 in 应该是指向不相同的内存的。

4）结束加密，输出最后的一段 512 字节的数据

```
EVP_EncryptFinal_ex(&ctx,out,&outl)
```

该函数会进行加密的检测，如果加密过程有误，一般会检查出来。

说明：解密跟上述过程是一样的，只不过要使用 EVP_Decrypt * ... * 系列函数。

以下通过具体实例说明上述过程。

5）编程实例——使用 Blowfish 算法加密一个字符串

```
/* ---  *  Description:一个测试例子用于对消息使用指定的算法进行加解密以及 base64 编码
 * ---  */
 #include <stdio.h>
 #include <string.h>
 #include <openssl/evp.h>
//base64 中每行的长度,最后一位是换行符号
#define CHARS_PER_LINE_BASE64 65   //64+1(\r)
void print(const char * promptStr,unsigned char * data,int len)
{
    int i;
    printf("====== %s[长度 = %d] ======\n",promptStr,len);
    for(i = 0; i < len; i++) printf("%02x", data[i]);
    printf("\n===============\n");
}
//base64 编码
```

```
void encode(unsigned char * outData,
            int * outlen,
            const unsigned char * data,
            int datalen)
{
    int tmp = 0;
    EVP_ENCODE_CTX base64;
    EVP_EncodeInit(&base64);                //base64 编码初始化
    //编码数据,由于数据不多,所以没有使用循环
    EVP_EncodeUpdate(&base64,               //base64 编码上下文对象
        outData,                            //编码后的数据
        outlen,                             //编码后的数据长度
        data,                               //要编码的数据
        datalen                             //要编码的数据长度
    );
    tmp = * outlen;
    //结束 base64 编码,事实上此时数据已经编码完成
    EVP_EncodeFinal(&base64,outData + * outlen,outlen);
    * outlen += tmp;
    outData[ * outlen] = 0;
    printf("base64 编码后:",outData, * outlen);
}

//base64 解码
bool decode(unsigned char * outData,
            int * outlen,
            const unsigned char * data,
            int datalen)
{
    int tmp = 0, i = 0, lines = 0, currpos = 0;
    EVP_ENCODE_CTX base64;
    EVP_DecodeInit(&base64);                //base64 解码初始化
    //假定 outData 缓冲区能够容纳所有的结果
    for (; ;)
{
      currpos += CHARS_PER_LINE_BASE64 * lines++;
    //下面函数的返回值中: i = 1 表示还有更多行需要解码
    //i = 0 表示没有进一步的数据需要解码
      i = EVP_DecodeUpdate(&base64,         //base64 解码上下文对象
        outData + tmp,                      //解码后的数据
        outlen,                             //解码后的数据长度
        data + currpos,                     //要解码的数据
        datalen - currpos);                 //要解码的数据长度
  if (i < 0)
    {
        printf("解码错误!\n");
        return false;
    }
    tmp += * outlen;
    if (i == 0) break;                      //数据结束
```

```
    }
    //结束 base64 解码
    EVP_DecodeFinal(&base64,outData + tmp,outlen);
    *outlen = tmp;
    outData[ *outlen] = 0;
    print("base64 解码后:",outData, *outlen);
    return true;
}
void main(int argc, char *argv[])
{
    const int ITERATIVE_ROUND_FOR_KEY = 3;
    unsigned char key[EVP_MAX_KEY_LENGTH];          //密钥
    unsigned char iv[EVP_MAX_IV_LENGTH];            //初始向量
    EVP_CIPHER_CTX ctx;                             //加密上下文对象
    unsigned char out[512 + 8];
    int outl;
    unsigned char txtAfterDecrypt[512];
    int txtLenAfterDecrypt;
    char simpleText[ ] = "Let's pray for peace of our lovely world";
    unsigned char txtAfterBase64[sizeof(simpleText) * 3];
    //密码
    const char *passwd = "a78b5C";                  //用于产生密钥的口令字符串
    const EVP_CIPHER *type;                         //加密类型对象
    OpenSSL_add_all_ciphers();                      //加载加密算法
    OpenSSL_add_all_digests();                      //加载摘要计算算法
    printf("密码是:%s\n",passwd);
    type = EVP_des_ede3_cbc();
    printf("密钥长度 = %d,向量长度 = %d\n",type->key_len,type->iv_len);
    //从文本密码中产生密钥/向量
    //这个例程使用 SHA-1 并且采用来自 RSA 的 PCKS#5 的标准
    EVP_BytesToKey(type,                            //密钥类型
        EVP_SHA-1( ),                               //摘要计算类型
        NULL,
        (const unsigned char *)passwd,              //口令串
        (int)strlen(passwd),                        //口令串长度
        ITERATIVE_ROUND_FOR_KEY,                    //迭代轮数
        key,                                        //输出的密钥
        iv                                          //输出的初始向量
        );
    //加密初始化,ctx 是加密上下文对象
    EVP_EncryptInit(&ctx,type,key,iv);
    int tmp = (int)strlen(simpleText);
    //由于数据量少,不用循环加入数据
    EVP_EncryptUpdate(&ctx,                         //加密上下文对象
        out,                                        //加密后的内容
        &outl,                                      //加密后的内容长度
        (const unsigned char *)simpleText,          //要加密的内容
        (int)strlen(simpleText)                     //要加密的内容长度
        );
    tmp = outl;
```

```
    //结束加密
    EVP_EncryptFinal(&ctx,out + outl,&outl);
    outl += tmp;
    //清除加密上下文,因为下文还要重用
    EVP_CIPHER_CTX_cleanup(&ctx);
    print("加密之后的结果: ",out,outl);
    //进行 base64 编码
    encode(txtAfterBase64,&tmp,out,outl);
    memset(out,0,sizeof(out));
    //进行 base64 解码
    decode(out,&outl,txtAfterBase64,tmp);
    //解密初始化,解密类型,密钥,初始向量必须和加密时相同,否则解密不能成功
    EVP_DecryptInit(&ctx,type,key,iv);
    EVP_DecryptUpdate(&ctx,                        //解密上下文对象
        txtAfterDecrypt,                           //解密后的内容
        &txtLenAfterDecrypt,                       //解密后的内容长度
        out,                                       //要解密的内容
        outl                                       //要解密的内容长度
        );
    tmp = txtLenAfterDecrypt;
    //结束解密
    EVP_DecryptFinal(&ctx,txtAfterDecrypt + txtLenAfterDecrypt,&txtLenAfterDecrypt);
    txtLenAfterDecrypt += tmp;
    EVP_CIPHER_CTX_cleanup(&ctx);
    txtAfterDecrypt[txtLenAfterDecrypt] = 0;
    printf("解密之后(长度 = %d):\n[%s]\n",txtLenAfterDecrypt,txtAfterDecrypt);
    printf("click any key to continue.");
    //相当于暂停,观察运行结果
    getchar();
}
```

运行结果如图 4-5 所示。



图 4-5　消息加解密及 base64 编码运行结果图

4.2.4　公钥算法编程

1. EVP_PKEY 系列函数

EVP API 对公开密钥算法的支持是通过 EVP_PKEY 系列的函数表现的。OpenSSL 公钥相关算法的 EVP 函数主要包括 RSA 密钥对生成函数、RSA 盲化函数、EVP 公钥加密

函数、EVP 公钥解密函数、RSA 公钥释放函数等，以下分别说明。

1）RSA 密钥对生成函数

```
RSA   * RSA_generate_key(
          int   num,
          unsigned long e,
          void ( * callback)(int, int , void * ),
          void ( * cb_arg)
```

功能：生成 RSA 密钥对。

返回值：成功生成新的 RSA 对象，失败返回 NULL。

参数说明如下。

num[in]：模数的 bit 位数。

e[in]：公钥。

callback[in]：用于报告素数生成状态的回调函数，如果不指定可以为 NULL。

cb_arg[in]：指向应用程序相关的数。如果 callback 函数为空，此参数无意义。

2）RSA 盲化函数

由于 RSA 在不同的私钥长度下的加解密时间不同，使得对 RSA 存在一种"时间测度攻击"，就是通过测量 RSA 解密速度来猜测其私钥。盲化是为抵御这种形式的攻击，使得不同私钥长度下的加解密时间具有几乎相同的时间特征。

```
int RSA_blinding_on(
      RSA    * rsa,
      BN_CTX    * ctx
    );
```

功能：RSA 盲化。

返回值：失败返回 0，成功返回非 0。

参数说明如下。

rsa[in]：RSA 对象。

ctx[in]：盲化时指定的一个大数，如果为空的话，系统自动分配一个新的大数，相当于一个均衡因子。

3）EVP 公钥加密函数

```
   int  EVP_PKEY_encrypt
(EVP_PKEY_CTX * ctx,
     unsigned char * out,
   size_t * outlen,
   const unsigned char * in,
   size_t inlen
   );
```

功能：EVP 公钥加密。

返回值：失败返回 1，成功返回加密后的数据长度。

参数说明如下。

ctx[in]：公钥对象。

out[in]：加密后的密钥。

outlen：加密后的密钥长度。

in：需要加密的密钥。

inlen：需要加密的密钥长度。

4）EVP 公钥解密函数

```
int  EVP_PKEY_decrypt(
  EVP_PKEY_CTX *ctx,
  unsigned char *out,
  size_t *outlen,
  const unsigned char *in,
  size_t inlen
);
```

功能：EVP 公钥解密。

返回值：失败返回 1，成功返回解密后的数据长度。

参数说明如下。

ctx[in]：私钥对象。

out[in]：解密后的密钥。

outlen：解密后的密钥长度。

in：需要解密的密钥。

inlen：需要解密的密钥长度。

5）RSA 公钥释放函数

```
void   EVP_PKEY_free(
       EVP_PKEY   *pkey
       );
```

功能：释放在公钥对象中所分配的内存。

参数说明如下。

pkey[in,out]：要被释放的公钥对象。

2. 应用举例

```
    /* ----- *  Description:  一个例子用于实现基于 OpenSSL 的 RSA 算法以及使用其进行
数据加解密操作  * ----- */
#include <stdio.h>
#include <string.h>
#include <openssl/evp.h>
#include <openssl/rsa.h>
void print(const char *promptStr, char *data, int len)
{
    int i;
    printf("\n=== %s[长度 = %d 字节] ======\n", promptStr, len);
    for (i = 0; i < len; i++) printf("%02x", data[i]);
    printf("\n======================\n");
}
```

```
//作为一条规则,使用静态调用
static void prime_generate_status(int code, int arg, void * cb_arg)
{
    if (arg>0 && (arg % 10))
    {
        printf(".");                    //每 5 个测试显示一个点以表示测试进度
        return;
    }
    if (code == 0)
        printf("\n 找到潜在素数: # %d#", (arg + 1));
    else if (code != 1)
        printf("\n 成功获取一个素数!");
}
//如果成功返回包装了 RSA 参数的 EVP_PKEY,否则返回 NULL
EVP_PKEY * getRSA()
{
    EVP_PKEY * pkey = NULL;
    RSA * rsa = RSA_generate_key(1024, //公钥模长
        RSA_3,                          //第三个费尔玛数作为公钥中的 e
        prime_generate_status,          //素数产生状态的回调函数
        NULL                            //传给回调函数的参数
        );
    if (NULL == rsa)
    {
        printf("生成 RSA 密钥对失败\n");
        return NULL;
    }
    //隐藏 RSA 密钥抵御定时攻击
    RSA_blinding_on(rsa, NULL);
    printf("\n 成功生成 RSA 密钥对\n");
    pkey = EVP_PKEY_new();
    if (NULL == pkey)
    {
        printf("EVP_PKEY_new failed\n");
        RSA_free(rsa);
        return NULL;
    }
    //将 RSA 对象赋给 EVP_PKEY 结构
    EVP_PKEY_assign_RSA(pkey, rsa);
    return pkey;
}
void main(int argc, char * argv[])
{
    RSA * rsa = NULL;
    EVP_PKEY * pkey = NULL;
    int len = -1;
    //要加密的明文
    unsigned char plainText[] = "[For test to public/private key encryption/decryption]";
    unsigned char encData[512];         //加密后的数据
    unsigned char decData[512];         //解密后的数据,应该和明文相同
```

```
OpenSSL_add_all_ciphers();
pkey = getRSA();
if (pkey == NULL)
{
    exit( - 1);
}
//加密
EVP_PKEY_CTX * ctx = NULL;
//从盲化密钥中提取公钥
ctx = EVP_PKEY_CTX_new(pkey, NULL);
if (NULL == ctx)
{
    printf("ras_pubkey_encryptfailed to open ctx.\n");
    EVP_PKEY_free(pkey);
    exit( - 1);
}
//初始化
if (EVP_PKEY_encrypt_init(ctx) <= 0)
{
    printf("ras_pubkey_encryptfailed to EVP_PKEY_encrypt_init.\n");
    EVP_PKEY_free(pkey);
    exit( - 1);
}
size_t enc_LENGTH;
len = EVP_PKEY_encrypt(
    ctx
    ,//公钥
    encData,                //加密后的数据
    &enc_LENGTH,            //密文长度
    plainText,              //明文
    sizeof(plainText)       //明文长度
    );
if (len == -1)
{
    printf("EVP_PKEY_encrypt 加密失败\n");
    exit( - 1);
}

print("加密后的数据", (char * )encData, enc_LENGTH);
//解密
//从盲化密钥中提取私钥
ctx = EVP_PKEY_CTX_new(pkey, NULL);
if (NULL == ctx)
{
    printf("ras_prikey_decryptfailed to open ctx.\n");
    EVP_PKEY_free(pkey);
    exit( -1);
}
//初始化
if (EVP_PKEY_decrypt_init(ctx) <= 0)
```

```
        {
            printf("ras_prikey_decryptfailed to EVP_PKEY_decrypt_init.\n");
            EVP_PKEY_free(pkey);
            exit ( - 1);
        }
        size_t plain_LENGTH;
        len = EVP_PKEY_decrypt(
            ctx,                        //公钥
            decData,                    //解密后的数据
            &plain_LENGTH,              //明文长度
            encData,                    //密文
            enc_LENGTH                  //密文长度
            );
        if (len ==  - 1)
        {
            printf("EVP_PKEY_decrypt 解密失败\n");
            exit( - 1);
        }
        printf("解密后的数据: % s", decData);
        printf("\n 明文是:[长度 = % d 字节]: % s\n",  plain_LENGTH, decData);
        //释放 EVP_PKEY 对象,其关联的 RSA 对象也被同时释放
        EVP_PKEY_free(pkey);
        printf("\n click any key to continue.");
        getchar();
    }
        }
```

运行结果如图 4-6 所示。

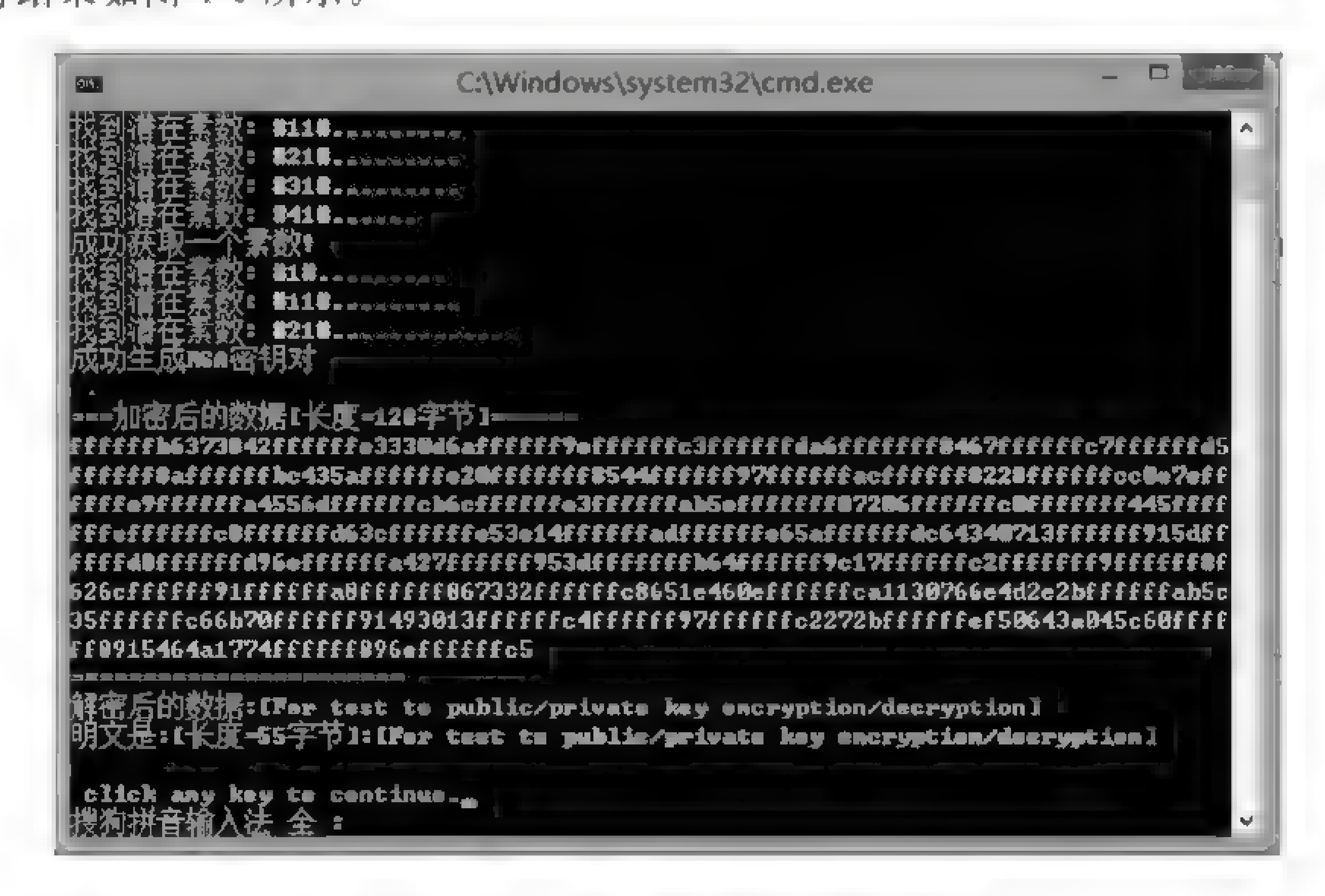


图 4-6 OpenSSL 公钥加密运行结果图

4.2.5　哈希摘要算法

哈希摘要系列函数封装了 OpenSSL 加密库所有的信息摘要算法，通过这种 EVP 封装，当使用不同的信息摘要算法时，只需要对初始化参数修改一下就可以了，其他代码可以完全一样。这些算法包括 MD2、SHA-1 以及 SHA 等算法。

1. 主要数据结构

1）EVP_MD 结构介绍

所有算法都维护着下面定义的结构的一个指针，在此基础上实现了算法的功能。EVP_MD 结构如下定义。

```
typedef struct env_md_st
{
  int type;
  int pkey_type;
  int md_size;
  unsigned long flags;
  int ( * init)(EVP_MD_CTX * ctx);
  int ( * update)(EVP_MD_CTX * ctx,const void * data,size_t count);
  int ( * final)(EVP_MD_CTX * ctx,unsigned char * md);
  int ( * copy)(EVP_MD_CTX * to,const EVP_MD_CTX * from);
 int ( * cleanup)(EVP_MD_CTX * ctx);
 int ( * sign)(int type, const unsigned char * m, unsigned int m_length,
 unsigned char * sigret, unsigned int * siglen, void * key);
 int ( * verify)(int type, const unsigned char * m, unsigned int m_length,
 const unsigned char * sigbuf, unsigned int siglen,
 void * key);
 int required_pkey_type[5];
 int block_size;
 int ctx_size;/ * how big does the ctx-> md_data need to be * /
  } EVP_MD;
```

功能：该结构用来存放摘要算法信息、非对称算法类型以及各种计算函数。

参数说明如下。

type：信息摘要算法的 NID 标识。

pkey_type：信息摘要-签名算法体制的相应 NID 标识，如 NID_shaWithRSAEncryption。

md_size：信息摘要算法生成的信息摘要的长度，如 SHA 算法是 SHA_DIGEST_LENGTH，该值是 20。

init：指向一个特定信息摘要算法的初始化函数，如对于 SHA 算法，指针指向 SHA_Init。

update：指向一个真正计算摘要值的函数，例如 SHA 算法就是指向 SHA_Update。

final：指向一个信息摘要值计算之后要调用的函数，该函数完成最后的一块数据的处理工作。例如，SHA 算法就是指向 SHA_Final。

copy：指向一个可以在两个 EVP_MD_CTX 结构之间复制参数值的函数。

required_pkey_type：指向一个用来签名的算法 EVP_PKEY 的类型，如 SHA 算法就指向 EVP_PKEY_RSA_method。

block_size：一个用来进行信息摘要的输入块的长度(单位是字节)，如 SHA 算法就是

SHA_CBLOCK。

ctx_size：CTX 结构的长度，在 SHA 算法里面应该就是 sizeof(EVP_MD *) + sizeof(SHA_CTX)。

如果要增加新的算法，那么可以定义这个结构，并进行必要的赋值，然后就可以使用通用的函数了。跟 EVP_CIPHER 系列函数一样，使用这个封装技术，就可以在使用一种摘要算法时，比如 SHA-1，在连接程序的时候就只连接 SHA-1 的代码。如果使用证书来标识算法，那么就会导致所有其他的信息摘要算法代码都连接到程序中去了。

2) EVP_MD_CTX 结构介绍

在调用函数的时候，一般来说需要传入 type 参数和 EVP_MD_CTX 结构，其定义如下。

```
typedef struct env_md_ctx_st
{
 const EVP_MD *digest;
   ENGINE *engine;
   unsigned long flags;
   void *md_data;
}EVP_MD_CTX;
```

参数说明如下。

digest：指向上面介绍的 EVP_MD 结构的指针。

engine：如果算法由 ENGINE 提供，该指针指向该 ENGINE。

md_data：信息摘要数据。

2. OpenSSL 支持的摘要算法

OpenSSL 对于各种摘要算法实现了上述结构，通过这些结构封装了各个摘要相关的运算。支持的信息摘要算法包括：

```
EVP_md_null(void)
EVP_md2(void)
EVP_md4(void)
EVP_SHA-1(void)
EVP_sha(void)
EVP_sha1(void)
EVP_dss(void)
EVP_dss1(void)
EVP_mdc2(void)
EVP_ripemd160(void)
```

3. 相关函数说明

EVP API 对于哈希摘要算法编程的支持，提供的主要函数包括：摘要上下文初始化函数，摘要计算初始化函数以及摘要计算更新函数和摘要计算结束函数。具体函数如下。

1) 摘要上下文初始化函数

```
int EVP_MD_CTX_init(
  EVP_MD_CTX *ctx
);
```

功能：初始化上下文对象。

返回值：失败返回 0，成功返回非 0 值。

参数说明如下。

ctx[in,out]：EVP 摘要对象。

2）摘要计算初始化函数

```
 int EVP_Digestinit(
 EVP_MD_CTX  * ctx,
   EVP_MD  md
);
```

功能：初始化摘要计算。

返回值：失败返回 0，成功返回非 0 值。

参数说明如下。

ctx[in,out]：EVP 摘要对象。

md[in]：EVP 摘要算法。

3）摘要计算更新函数——用于不断加入要被计算摘要的数据

```
int  EVP_DigestUpdate(
   EVP_MD_CTX  * ctx,
   unsigned  char * in,
   int  inl
);
```

功能：摘要计算更新。

返回值：失败返回 0，成功返回非 0 值。

参数说明如下。

ctx[in,out]：EVP 摘要对象。

in[in]：输入数据(将要被计算摘要的)缓冲区。

inl[in]：输入数据的字节长度。

4）摘要计算结束函数——用于输出摘要值

```
int EVP_DigestFinal (
 EVP_MD_CTX   * ctx,
  unsigned  char  * out,
  int * outl
);
```

功能：摘要计算更新。

返回值：失败返回 0，成功返回非 0 值。

参数说明如下。

ctx[in,out]：EVP 摘要对象。

out[out]：存放摘要值的数据缓冲区。

outl[out]：存放摘要值的字节长度。

4. 应用举例

```
/* -- Description: 一个测试用指定的摘要计算方法计算消息摘要的例子程序 */
 #include <stdio.h>
#include <string.h>
#include <openssl/evp.h>
void main(int argc, char *argv[])
{
    EVP_MD_CTX mdctx;
    const EVP_MD *md;
    //要被计算的消息数组
    char msgs[][64] = { "It's just for test",
        "Author: Jian Shen", "Hello World from openssl" };
    //摘要算法名称
    const char *digestName = "sha1";   //"sha1"
    //const char *digestName = "sha1";
    unsigned char md_value[EVP_MAX_MD_SIZE];
    unsigned int  md_len, i;
    OpenSSL_add_all_digests();
    //依据摘要名称获取摘要算法对象
    md = EVP_get_digestbyname(digestName);
    if (!md) {
        printf("错误的摘要算法名称:%s\n", digestName);
        exit(1);
    }
    EVP_MD_CTX_init(&mdctx);
    //初始化摘要计算
    EVP_DigestInit(&mdctx, md);
    //循环加入要计算摘要的数据
    for (i = 0; i<sizeof(msgs) / sizeof(msgs[0]); i++)
    {
        EVP_DigestUpdate(&mdctx,         //摘要计算上下文
            msgs[i],                     //要加入的信息
            strlen(msgs[i])              //信息长度
            );
    }
    //结束摘要计算,并输出摘要值,md_len是摘要值的长度
    EVP_DigestFinal(&mdctx, md_value, &md_len);
    EVP_MD_CTX_cleanup(&mdctx);
    printf("摘要值是:(类型=%s,长度=%d字节):", digestName, md_len);
    for (i = 0; i<md_len; i++) printf("%02x", md_value[i]);
    printf("\n");
    printf("\n click any key to continue.");
    //相当于暂停,观察程序运行结果
    getchar();
}
```

运行结果图 4 7 所示。

```
摘要值是:(类型=sha1,长度=20字节): 70926e73aac49c17ed7705b0919424733581156?
```

图 4-7 OpenSSL 摘要计算运行结果图

4.2.6　消息鉴别码 HMAC

对于消息鉴别码的生成，OpenSSL 仅支持 HMAC 算法。HMAC 用于保护消息的完整性，它采用摘要算法对消息、填充以及秘密密钥进行混合运算。在消息传输时，用户不仅传送消息本身，还传送 HMAC 值。接收方接收数据后也进行 HMAC 运算，再比对 MAC 值是否一致。由于秘密密钥只有发送方和接收方才有，其他人不可能伪造假的 HMAC 值，从而能够知道消息是否被篡改。

1. 主要函数

HMAC 的实现在 crypto/hmac/hmac.c 中，具体函数如下。

```
  unsigned char * HMAC(const EVP_MD * evp_md, const void * key, int key_len, const unsigned
char * d, size_t n, unsigned char * md, unsigned int * md_len)
 {
  HMAC_CTX c;
  static unsigned char m[EVP_MAX_MD_SIZE];
  if (md == NULL) md = m;
    HMAC_CTX_init(&c);
    HMAC_Init(&c,key,key_len,evp_md);
    HMAC_Update(&c,d,n);
     HMAC_Final(&c,md,md_len);
    HMAC_CTX_cleanup(&c);
   return(md);
}
```

参数说明如下。

evp_md：指明 HMAC 使用的摘要算法。

key：秘密密钥指针地址。

key_len：秘密密钥的长度。

d：需要做 HMAC 运算的数据指针地址。

n：d 的长度。

md：用于存放 HMAC 值。

md_len：为 HMAC 值的长度。

2. 应用举例

```
/ * Description: a test example to do compute message authenticate code. —个测试用例以计算消
息验证码 * /
 #include <stdio.h>
 #include <string.h>
 #include <openssl/evp.h>
 #include <openssl/hmac.h>
//以十六进制打印无符号提示字符串
void print(const char * promptStr,unsigned char * data,int len)
{
    int i;
```

```
    if(promptStr!= NULL)
    {
        printf("====== %s[out len = %d] ======\n",promptStr,len);
    }
    for(i = 0; i < len; i++) printf("%02x", data[i]);
    if(promptStr!= NULL)
    {
        printf("\n===============\n");
    }
}
//以十六进制打印无符号提示字符串
void printDirectly(unsigned char *data,int len)
{
    print(NULL,data,len);
}
/* Return the actual lenght be read out.
   return 0 when the file is meet the end.
   and -1 when error occurs.
   @param len-- the actual lenght be read out.
   @parma buf -- the buffer to hold the content.
   @parma buflen-- the capacity of the buffer
*/
unsigned int read_file(FILE *f,unsigned char * buf, int buflen)
{
    int actualReadLen = 0;
    if(buflen<=0)
    {
        printf("缓冲区长度无效\n");
        return -1;
    }
    actualReadLen = (int)fread(buf, sizeof(unsigned char), buflen, f);
    if (actualReadLen > 0)
    {
       return actualReadLen;
    }
    if ( feof(f) )
    {
        return 0;
    }
    return -1;
}
//需从命令行中输入文件
void main(int argc, char *argv[])
{
    char key[] = "simple_key";
    const char* digest = "sha";
    const char* srcfile = "mac.c";           //tobe calculate mac
    FILE *f = NULL;
    EVP_MD_CTX mdctx;
    const EVP_MD *md;
```

```
    unsigned char mac_value[EVP_MAX_MD_SIZE];
    unsigned int mac_len = 0;
    HMAC_CTX mac_ctx;
    unsigned char buffer[2048];
    int actualReadLen = 0;
    OpenSSL_add_all_digests();
    md = EVP_get_digestbyname(digest);
    if(!md) {
        printf("不能识别的信息摘要: %s\n", digest);
        exit(1);
    }
    EVP_MD_CTX_init(&mdctx);
    EVP_DigestInit_ex(&mdctx, md, NULL);
    if( (f = fopen(srcfile, "rb")) == NULL )
    {
        printf("打开文件 %s 时失败\n", srcfile);
        exit(1);
    }
    HMAC_Init(&mac_ctx, key, sizeof(key), md);
        for(;;)
    {
        actualReadLen = read_file(f, buffer, sizeof(buffer));
        if( actualReadLen == 0) break;     //finish reading.
        if( actualReadLen < 0)
        {
            printf("错误发生在文件 [ %s]\n", srcfile);
            exit(1);
        }
        HMAC_Update(&mac_ctx, buffer, actualReadLen);
    }
    HMAC_Final(&mac_ctx, mac_value, &mac_len);
    HMAC_cleanup(&mac_ctx);
    printf("HMAC( %s, %s) = ", srcfile,key);
    printDirectly(mac_value, mac_len);
    printf("\n");
    printf("\n click any key to continue.");
    //相当于暂停,便于观察运行结果
    getchar( );
}
```

运行结果如图 4-8 所示。



图 4-8　OpenSSL HMAC 计算运行结果图

4.2.7 签名和验证算法

OpenSSL 中的验证是先对原始数据计算摘要，再对摘要进行私钥加密。验证的过程是对原始消息计算摘要、解密验证值，和摘要对比是否一致。如果一致，说明验证有效；否则，认为原文或验证值已经被篡改。

1. 主要函数

1）签名初始化函数

```
int EVP_SignInit_ex(
   EVP_MD_CTX   * ctx,
    const EVP_MD   md,
   ENGINE * impl);
```

功能：初始化签名。

返回值：失败返回 0，成功返回非 0 值。

参数说明如下。

ctx[in,out]：EVP 摘要对象。

md[in] ：EVP 摘要算法。

2）签名更新函数——用于不断加入要被计算签名的数据

```
int EVP_SignUpdate(
      EVP_MD_CTX * ctx,
        unsigned char   * in,
      int   inl
      );
```

功能：签名计算更新。

返回值：失败返回 0，成功返回非 0 值。

参数说明如下。

ctx[in,out]：EVP 摘要对象。

in[in] ：输入数据(将要被计算摘要的)缓冲区。

inl[in] ：输入数据的字节长度。

3）签名结束函数——用于输出签名值

```
int EVP_SignFinal(
  EVP_MD_CTX * ctx,
     unsigned char * out,
     int * outl,
    EVP_PKey pkey
    );
```

功能：计算签名结束，输出签名值。

返回值：失败返回 0，成功返回非 0 值。

参数说明如下。

ctx[in,out]：EVP 摘要对象。

out[out]：存放摘要值的数据缓冲区。

outl[out]：存放摘要值的字节长度。

pkey[in]：签名所用的私钥。

其中，私钥计算过程如下。

```
RSA rsa = RSA_generate_key(1024, RSA_F4, NULL, NULL); //产生一个 1024 位的 RSA 密钥
 EVP_PKEY * evpKey = EVP_PKEY_new( );                 //新建一个 EVP_PKEY 变量
EVP_PKEY_set1_RSA(evpKey, rsa);                       //保存 RSA 结构体到 EVP_PKEY 结构体
                                                      //完成任务就可以用 evpKey 来签名了
 EVP_PKey(pkey);
```

对于验证，其函数说明与上述签名过程相同，把函数名 EVP_Sign *** 变换为 EVP_Verify *** 即可。

2. 应用举例

```
 /*    一个使用 EVP 接口进行 DSA/RSA 签名和验证的例子 */
#include <stdio.h>
#include <string.h>
#include <openssl/evp.h>
#include <openssl/dsa.h>
#include <openssl/rsa.h>
#include <openssl/err.h>

void print(const char *promptStr,unsigned char *data,int len)
{
    int i;
    printf("\n=== %s[长度 = %d] ======\n",promptStr,len);
    for(i = 0; i < len; i++) printf("%02x", data[i]);
    printf("\n=======================\n");
}

void print_errors()
{
    /*
    int flags, line;
    char *data, *file;
    unsigned long code;
    code = ERR_get_error_line_data(&file, &line, &data, &flags);
    while (code)
```

```
    {
        printf("error code: %lu in %s line %d.\n", code, file, line);
        if (data && (flags & ERR_TXT_STRING))
            printf("error data: %s\n", data);
        code = ERR_get_error_line_data(&file, &line, &data, &flags);
    }
    */
}
//如果成功返回 EVP_PKEY,否则返回 NULL
EVP_PKEY * getDSA()
{
    EVP_PKEY * pkey = NULL;
    int ret = 0;
    DSA * dsa = DSA_generate_parameters(1024,NULL,0,NULL,NULL,NULL,NULL);
    if(NULL == dsa)
    {
        printf("生成 DSA 参数失败\n");
        return NULL;
    }
printf("\nDSA 参数成功产生\n");
    ret = DSA_generate_key(dsa);
    if(ret == 0)
    {
        printf("DSA_generate_key 失败\n");
        goto err;
    }
printf("\nDSA 钥对成功产生\n");
    pkey = EVP_PKEY_new();
    if(NULL == pkey)
    {
        printf("EVP_PKEY_new 失败\n");
        goto err;
    }
    EVP_PKEY_assign_DSA(pkey,dsa);
    return pkey;
err:
    DSA_free(dsa);
    return NULL;
}
//如果成功返回 EVP_PKEY,否则返回 NULL
EVP_PKEY * getRSA()
{
    EVP_PKEY * pkey = NULL;
    RSA * rsa = RSA_generate_key(1024,RSA_3,NULL,NULL);
    if(NULL == rsa)
    {
        printf("生成 RSA 密钥对失败\n");
        return NULL;
```

```
    }
// 盲化 RSA 密钥以抵御时间攻击
    RSA_blinding_on(rsa,NULL);
    printf("\n 生成 RSA 密钥对成功\n");
    pkey = EVP_PKEY_new();
    if(NULL == pkey)
    {
        printf("EVP_PKEY_new 失败\n");
        RSA_free(rsa);
        return NULL;
    }
    EVP_PKEY_assign_RSA(pkey,rsa);
    return pkey;
}

void main(int argc, char * argv[])
{
    EVP_MD_CTX mdctx;
    const EVP_MD * md;
    char msgs[ ][64] = {"It's just for test",
        "Author: Jian Shen","Hello World from openssl"};
//确切地说,应该从 EVP_PKEY_size()获得
//应该注意不要使用 EVP_MAX_MD_SIZE
    unsigned char sig_value[1024];
    char mdName[ ] = "dss1";              //dss1 是唯一被 DSA 支持的信息摘要算法
    //char mdName[ ] = "sha1";
        unsigned int sig_len;
    int i;
    DSA * dsa = NULL;
    EVP_PKEY * pkey = NULL;
    OpenSSL_add_all_digests();
    //不要遗漏下面的步骤否则 EVP_SignFinal 会出错!!
    OpenSSL_add_all_ciphers();
    md = EVP_get_digestbyname(mdName);
    if(!md) {
        printf("错误的摘要算法名称 % s\n", mdName);
        exit(1);
    }
    pkey = getDSA();                      //获取 DSA 对象
    //pkey = getRSA();
    if(pkey == NULL)
    {
        exit( - 1);
    }
EVP_MD_CTX_init(&mdctx);                  //初始化摘要计算上下文
    EVP_SignInit(&mdctx, md);             //初始化签名算法
                                          //EVP_SignInit_ex(&mdctx, md,NULL);
    for( i = 0;i < sizeof(msgs)/sizeof(msgs[0]);i++)
```

```
    {
        EVP_SignUpdate(&mdctx, msgs[i], strlen(msgs[i]));
    }
//计算签名：实际上该步先计算出摘要值，然后对该值用私钥签名
    i = EVP_SignFinal(
    &mdctx,                                             //摘要计算上下文对象
    sig_value,                                          //输出的签名值
    &sig_len,                                           //签名值长度
    pkey //签名所用私钥
    );
    //i = EVP_DigestFinal(&mdctx,sig_value,&sig_len);   //successfully
    if(i == 0)
    {
        printf("计算签名失败");
        exit(1);
    }
    EVP_MD_CTX_cleanup(&mdctx);                         //释放摘要上下文对象
    EVP_PKEY_free(pkey);                                //释放公钥对象
    print("签名值是：",sig_value,sig_len);
    printf("\n click any key to continue.");
//相当于"暂停",以便观察运行结果
    getchar();
}
cksum = (cksum >> 16) + (cksum &0xffff);
cksum += (cksum >> 16);
return (USHORT)( - cksum);
}
//cksum = (cksum >> 16) + (cksum &0xffff);
//cksum += (cksum >> 16);
//return (USHORT)( - cksum);
//}
```

运行结果如图 4-9 所示。



图 4-9 OpenSSL 签名运行结果图

小 结

OpenSSL 作为应用广泛的网络安全开发包，基于 OpenSSL EVP 编程能够实现对称加密、公钥加密以及消息摘要、签名和验证等功能。

思　考　题

1. 基于 OpenSSL 开发包实现以下功能。

(1) 文件完整性验证。

(2) 基于 RSA 的数字签名。

2. 基于 OpenSSL 开发包设计并实现基于口令的身份识别。

3. 基于 OpenSSL 开发包实现本机上的基于 DES 的加密文件传输。

第5章　网络扫描器设计

网络安全扫描是指对计算机网络系统进行相关的安全检测，进而找出安全隐患和漏洞，帮助管理员发现系统的弱点并加以修补，有效避免非法入侵行为，防患于未然。

利用安全扫描技术编写的软件系统通常被称为网络安全扫描器，扫描器并不是可直接攻击网络的程序，它仅仅能让用户通过扫描得到的数据发现目标主机的某些内在的弱点。通过端口扫描，不仅可以发现目标主机的开放端口和操作系统的类型，还可以查找系统的安全漏洞，获得口令缺陷等相关信息，因此，掌握端口扫描基本工作原理与软件设计方法是信息安全专业人员必须掌握的基本技能之一。同时，研究网络端口扫描器的实现方法，对于维护网络系统的安全，了解黑客攻击的手段有着重要的意义。因此，本章接下来将针对网络扫描器的设计与具体实现进行详细的阐述。

5.1　基本知识

安全扫描技术是网络安全中的重要技术之一。安全扫描技术与防火墙、安全监控系统互相配合能够为网络提供很高的安全性。安全扫描工具起源于Hacker在入侵网络系统时所采用的工具。商品化的安全扫描工具为网络安全漏洞的发现提供了强大的支持。安全扫描工具通常也分为基于服务器的扫描器和基于网络的扫描器。

基于服务器的扫描器主要扫描服务器相关的安全漏洞，如password文件、目录和文件权限、共享文件系统、敏感服务、软件、系统漏洞等，并给出相应的解决办法与建议。

基于网络的安全扫描主要扫描设定网络内的服务器、路由器、网桥、交换机、访问服务器、防火墙等设备的安全漏洞，并可设定模拟攻击，以测试系统的防御能力。通常该类扫描器通过设定IP地址或路由器跳数限制使用范围。网络安全扫描的主要性能应该考虑以下几方面：

(1) 速度。在网络内进行安全扫描非常耗时。

(2) 网络拓扑。通过GUI的图形界面，可选择一部分或某些区域的设备。

(3) 能够发现的漏洞数量。

(4) 是否支持可定制的攻击方法。通常提供强大的工具来构造特定的攻击方法，由于网络内服务器及其他设备对相同协议的实现存在差别，预制的扫描方法无法满足客户的需求。

(5) 报告。扫描器应该能够给出清楚的安全漏洞报告。

(6) 更新周期。提供该项产品的厂商应尽快给出新发现的安全漏洞扫描特性升级，并给出相应的改进建议。

值得注意的是，安全扫描器不能实时监视网络上的入侵，但是能够测试和评价系统的安全性，并及时发现安全漏洞。

为了更好地了解安全扫描过程与具体实现，首先讲述以下相关的基本概念。

5.1.1 端口

1. 端口的含义

"端口"是英文Port的意译，可以认为是设备与外界通信交流的出口。端口可分为虚拟端口和物理端口，其中虚拟端口指计算机内部或交换机、路由器内的端口，是不可见的。例如，计算机中的80端口、21端口、23端口等。物理端口又称为接口，是可见端口，计算机背板的RJ-45网口，交换机、路由器、集线器等的RJ-45端口。电话使用的RJ-11插口也属于物理端口的范畴。在网络安全程序设计中，重点关注网络端口。

在网络技术中，端口有多种含义，集线器、交换机、路由器的端口指的是连接其他网络设备的接口，如RJ-45端口、Serial端口等。这里所指的端口不是指物理意义上的端口，而是特指TCP/IP中的端口，是逻辑意义上的端口。

在Internet上，各主机间通过TCP/IP进行数据包的发送和接收，各个数据包根据其目标主机的IP地址进行互联网络中的路由选择。可见，把数据包顺利地传送到目标主机是没有问题的。但是，大多数操作系统都支持多程序（进程）同时运行，那么目的主机应该把接收到的数据包传送给众多同时运行的进程中的哪一个呢？显然这个问题有待解决，端口机制便由此被引入进来。

本地操作系统会给那些有需求的进程分配协议端口（Protocol Port，即人们常说的端口），每个协议端口由一个正整数标识，如80、139、445等。当目标主机接收到数据包后，将根据报文首部的目标端口号，把数据发送到相应端口，而与此端口相对应的那个进程将会领取数据并等待下一组数据的到来。

端口其实就是队，操作系统为各个进程分配了不同的队，数据包按照目标端口被推入相应的队中，等待被进程取用。在极特殊的情况下，这个队也是有可能溢出的，不过操作系统允许各进程指定和调整自己的队的大小。

不光接收数据包的进程需要开启它自己的端口，发送数据包的进程也需要开启端口，因此，数据包中将会有源端口的标识，以便接收方能顺利地将数据包回传到这个端口。

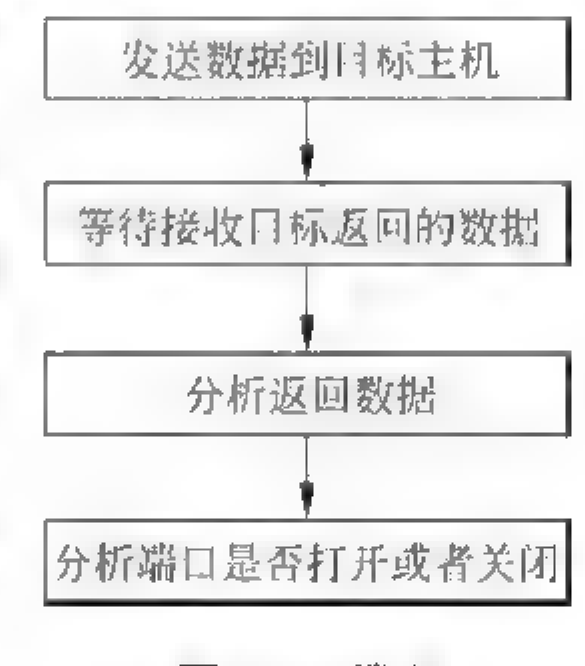


图5-1 端口

端口的示意如图5-1所示。

2. 端口的作用

众所周知，一台拥有IP地址的主机可以提供许多服务，如Web服务、FTP服务、SMTP服务等，这些服务都是对应于同一个IP地址。那么，主机是怎样区分不同的网络服务的呢？显然不能只靠IP地址，实际上是通过"IP地址+端口号"区分不同的服务的。

需要注意的是，端口号并不是一一对应的。比如客户的计算机作为客户机访问一台WWW服务器时，WWW服务器使用80端口与客户计算机通信，但客户计算机则可能使用3457这样的端口。

3. 端口分类

首先了解面向连接和无连接的协议（Connection-Oriented and Connectionless Protocols）。

面向连接的服务要经过三个阶段：建立连接、传输数据、释放连接。数据传输前，先建立连接，连接建立后再传输数据，数据传送完后，释放连接。面向连接服务可确保数据传送的秩序和传输的可靠性。而无连接服务只有传输数据阶段，消除了除数据通信外的其他开销。在无连接的服务中，只要发送实体是活跃的，接收实体无须活跃。它的优点是灵活方便、迅速，特别适合于传送少量零星的报文，但无连接服务不能防止报文的丢失、重复或乱序。

其次，区分"面向连接服务"和"无连接服务"的概念。面向连接服务和无连接服务之间的区分可以用形象的打电话和写信这两种方式进行比较。如果两个人要通电话，必须先建立连接，即拨号的过程，等待对方应答后才能相互传递信息，最后还要释放连接，即挂电话。写信就没有那么复杂了，将收信人地址姓名填好以后直接往邮筒一扔，收信人就能收到。TCP/IP在网络层是无连接的，数据包只管往网上发，如何传输和到达以及是否到达由网络设备管理。而"端口"是传输层的内容，是面向连接的。协议里面低于1024的端口都有确切的定义，它们对应着因特网上常见的一些知名服务。

根据提供服务类型的不同，端口分为两种，一种是TCP端口，另一种是UDP端口。计算机之间相互通信的时候，分为两种方式：一种是发送信息以后，可以确认信息是否到达，即有应答的方式，这种方式大多采用TCP；一种是发送以后就不管了，不去确认信息是否到达，这种方式大多采用UDP。对应这两种协议的服务提供的端口，也就分为TCP端口和UDP端口。

1）TCP端口

TCP是Transmission Control Protocol（传输控制协议）的缩写，是一种面向连接（连接导向）的、可靠的、基于字节流的传输层（Transport Layer）的通信协议，由IETF的RFC 793进行说明。在简化的计算机网络OSI模型中，它完成第4层传输层所指定的功能，UDP是该层内另一个重要的传输协议。

2）UDP端口

UDP是OSI参考模型中无连接的传输层协议，提供面向事务的简单不可靠信息传送服务。UDP基本上是IP协议与上层协议的接口。UDP适用端口分别运行在同一台设备上的多个应用程序。

网络中可以被命名和寻址的通信端口是操作系统的一种可分配资源。由网络OSI（Open System Interconnection Reference Model，开放系统互连参考模型）7层协议可知，传输层与网络层最大的区别是传输层提供进程通信能力，网络通信的最终地址不仅包括主机地址，还包括可描述进程的某种标识。所以TCP/IP提出的协议端口，可以认为是网络通信进程的一种标识符。

应用程序（调入内存运行后一般被称为进程）通过系统调用与某端口建立绑定后，传输层传给该端口的数据都被对应的进程所接收，对应进程发给传输层的数据都从该端口输出。在TCP/IP的实现中，端口操作类似于一般的I/O操作，进程获取一个端口，相当于获取本地唯一的I/O文件，可以用一般的读写方式访问类似于文件描述符，每个端口都拥有一个叫端口号的整数描述符，用来区别不同的端口。由于TCP/IP传输层的TCP和UDP两个协议是完全独立的软件模块，因此各自的端口号也相互独立。如TCP有一个255号端口，UDP也可以有一个255号端口，两者并不冲突。端口号有两种基本分配方式，第一种叫全局分配，这是一种集中分配方式，由一个公认权威的中央机构根据用户需要进行统一分配，并将结果公布于众；第二种是本地分配，又称动态连接，即进程需要访问传输层服务时，向

本地操作系统提出申请，操作系统返回本地唯一的端口号，进程再通过合适的系统调用，将自己和该端口绑定起来。TCP/IP 端口号的分配综合了以上两种方式，将端口号分为两部分，少量的作为保留端口，以全局方式分配给服务进程。每一个标准服务器都拥有一个全局公认的端口叫周知端口，即使在不同的机器上，其端口号也相同。剩余的为自由端口，以本地方式进行分配。TCP 和 UDP 规定，小于 256 的端口才能作为保留端口。

4. 协议端口

如果把 IP 地址比作一间房子，端口就是出入这间房子的门。真正的房子只有几个门，但是一个 IP 地址的端口可以有 65 536(即：2^{16})个之多。端口是通过端口号来标记的，端口号只有整数，范围是 0～65 535($2^{16}-1$)。

(1) 公认端口(Well-known Ports)：从 0 到 1023，它们紧密绑定于一些服务。通常这些端口明确表明了某种服务的协议。其中，80 端口分配给 WWW 服务，21 端口分配给 FTP 服务等。因此，用户在浏览器的地址栏里输入某个网址的时候不必指定端口号，因为在默认情况下 WWW 服务的端口就是 80。

(2) 注册端口(Registered Ports)：从 1024 到 49151。它们松散地绑定于一些服务。也就是说有许多服务绑定于这些端口，这些端口同样用于许多其他目的。例如，许多系统处理动态端口从 1024 左右开始。注册分配给用户进程或应用程序。这些进程主要是用户选择安装的一些应用程序，而不是已经分配好了公认端口的常用程序。这些端口在没有被服务器资源占用的时候，可以在用户端动态选用为源端口。

(3) 动态和/或私有端口(Dynamic and/or Private Ports)：从 49 152 到 65 535。之所以称为动态端口，是因为它一般不固定分配给某种服务，而是动态地分配。理论上，不应为服务分配这些端口。实际上，机器通常从 1024 起分配动态端口。但也有例外，比如 Sun 的 RPC 端口从 32 768 开始。

5. 复位向端口

一种常见的技术是把一个端口复位向到另一个地址。例如，默认的 HTTP 端口是 80，不少人将它复位向到另一个端口，如 8080。因为如果有人要对一个公认的默认端口进行攻击，则必须先进行端口扫描，而实现复位向是为了隐藏公认的默认端口，降低受破坏率。大多数端口复位向与原端口有相似之处，例如，多数 HTTP 端口由 80 变化而来：81，88，8000，8080，8888。同样，POP 的端口原来在 110，也常被复位向到 1100。也有不少情况是选取统计上有特别意义的数，如 1234、23 456、34 567 等。当然，也有许多人依据其他原因选择奇怪的数，如 42、69、666、31 337。越来越多的远程控制木马(Remote Access Trojans，RATs)采用相同的默认端口，如 NetBus 的默认端口是 12 345。BlakeR. Swopes 指出使用复位向端口还有一个原因，比如在 UNIX 系统上，如果想侦听 1024 以下的端口需要有 root 权限，那么如果没有 root 权限而又想开 Web 服务，就需要将其安装在较高的端口。此外，一些 ISP 的防火墙将阻挡低端口的通信，因此即使拥有整个机器用户还是需要复位向端口。

6. 端口在入侵中的作用

有人曾经把服务器比作房子，而把端口比作通向不同房间(即服务)的门，如果不考虑细节的话，这是一个不错的比喻。入侵者要占领这间房子，势必要破门而入(物理入侵另说)，那么对于入侵者来说，了解房子开了几扇门，都是什么样的门，门后面有什么东西就显得至

关重要。

入侵者通常会用扫描器对目标主机的端口进行扫描，以确定哪些端口是开放的，根据开放的端口，入侵者可以知道目标主机大致提供了哪些服务，进而猜测可能存在的漏洞，因此利用对端口的扫描可以更好地了解目标主机。而对于计算机管理员，扫描本机的开放端口也是做好安全防范的第一步。

那么，如果攻击者使用软件扫描目标计算机，得到目标计算机打开的端口，也就了解了目标计算机提供了哪些服务。众所周知，提供服务就一定有服务软件的漏洞，根据这些，攻击者可以达到对目标计算机的初步了解。假如计算机的端口打开太多，而管理者不知道，那么有两种可能的原因：一种是机器提供了服务而管理者没有注意，比如安装 IIS 的时候，软件就会自动增加很多服务，而管理员可能没有注意到；一种是服务器被攻击者安装了木马，通过特殊的端口进行通信。这两种情况都是很危险的，说到底，就是管理员不了解服务器提供的服务，降低了系统安全系数。

7. 相关工具

1) netstat /an

netstat 并不是一个工具，但这是查看自身所开放端口的最方便的方法，在 cmd 中输入这个命令就可以达到目的，如下。

```
C:\>netstat /an
Active Connections
Proto Local Address F
Foreign Address State
TCP 0.0.0.0:135 0.0.0.0:0 LISTENING
TCP 0.0.0.0:445 0.0.0.0:0 LISTENING
TCP 0.0.0.0:1025 0.0.0.0:0 LISTENING
TCP 0.0.0.0:1026 0.0.0.0:0 LISTENING
TCP 0.0.0.0:1028 0.0.0.0:0 LISTENING
TCP 0.0.0.0:3372 0.0.0.0:0 LISTENING
UDP 0.0.0.0:135 * : *
UDP 0.0.0.0:445 * : *
UDP 0.0.0.0:1027 * : *
UDP127.0.0.1:1029 * : *
UDP 127.0.0.1:1030 * : *
```

这是用户没上网的时候机器所开的端口，135 号和 445 号端口是固定端口，其余几个都是动态端口。

2) fport.exe 和 mport.exe

这是两个通过命令行查看本地机器开放端口的小程序，其实与 netstat /an 这个命令大同小异，只不过它能够显示打开端口的进程，信息更多一些而已，如果用户怀疑自己的奇怪端口可能是木马，就可以用它们进行检查。

3) activeport.exe(也称 aports.exe)

还是用来查看本地机器开放端口的程序，除了具有上面两个程序的全部功能外，它还有两个更吸引人的功能，分别是图形界面及可以关闭端口。这对初级用户来说是一个非常好用的工具。

4) SuperScan3.0

这是纯端口扫描类软件中的佼佼者，速度快而且可以指定扫描的端口，绝对是必备的工具。

5) Visual Sniffer

这个工具可以拦截网络数据包，查看正在开放的各个端口，非常好用。

8. 保护端口

刚接触网络的用户一般都对自己的端口很敏感，总怕自己的计算机开放了过多端口，更怕其中就有后门程序的端口，但由于对端口不是很熟悉，所以也没有解决办法，上起网来提心吊胆。其实保护自己的端口并不是那么难，只要做好下面几点就行了。

(1) 查看：经常用命令或软件查看本地所开放的端口，看是否有可疑端口。

(2) 判断：如果开放端口中有不熟悉的，应该马上查找端口大全或木马常见端口等资料，看看里面对那个可疑端口的作用描述，或者通过软件查看开启此端口的进程进行判断。

(3) 关闭：如果真是木马端口或者资料中没有这个端口的描述，那么应该关闭此端口，可以用防火墙来屏蔽此端口，也可以用"本地连接"→TCP IP→"高级"→"选项"→TCP/IP 筛选，启用筛选机制筛选端口。

需要注意的是，判断的时候要慎重，因为一些动态分配的端口也容易引起多余的怀疑，这类端口一般比较低，并且是连续的。还有一些狡猾的后门软件会借用 80 等一些常见端口进行通信(穿透了防火墙)，令人防不胜防，因此不要轻易运行陌生程序才是关键。

5.1.2 端口扫描

端口扫描通常指用同一个信息对目标主机所有需要扫描的端口进行探测数据的发送，然后根据返回数据的状态分析目标主机端口是否打开、是否可用。端口扫描首先与目标主机的端口建立连接并请求某些服务，判断目标主机的应答，通过所收集的目标主机的端口信息，确定端口所进行的服务类型以及服务的详细信息，发现目标主机的漏洞和弱点。端口扫描也可以通过捕获本地主机或服务器的流入流出数据包监视本地 IP 主机的运行情况，它能对接收到的数据进行分析，帮助发现目标主机的某些内在的弱点。

端口扫描是最基本的网络安全扫描技术，它的基本流程如图 5-2 所示。

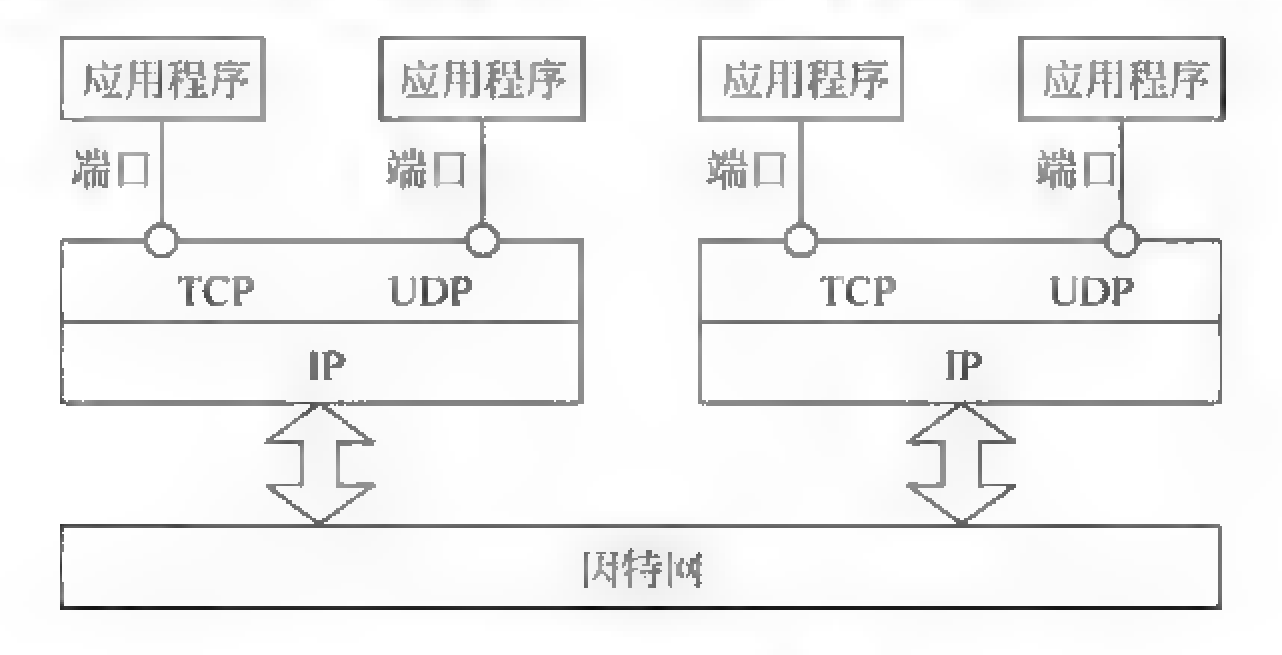


图 5-2 端口扫描流程

在这个流程中，发送数据是最重要的，而数据主要是根据不同的网络协议而构造的。如果是基于 TCP 的端口扫描，就可以利用 TCP 产生相应的数据。所以，此处数据指的是各种

不同的网络协议数据。

总的来说，端口扫描就是不断地对目标主机进行探测，但不同的方法有不同的探测技巧。根据TCP/IP，网络安全扫描主要分为几类，下面对这些方法分别进行讲述。

5.2 ICMP扫描

IP协议是一种不可靠的协议，无法进行差错控制，因此差错控制的功能就交给了ICMP(Internet Control Messages Protocol，Internet控制报文协议)，该协议允许主机或路由器报告差错情况并提供有关异常情况的报告。ICMP扫描是基本的扫描技术，人们所熟知的Ping程序是使用ICMP实现的，是一个完整的ICMP扫描案例。

5.2.1 ICMP协议

1. ICMP报文

ICMP经常被认为是IP层的一个组成部分。它传递差错信息及其他需要注意的信息给用户进程。ICMP报文通常被IP层或更高层协议(TCP或UDP)使用，ICMP报文是被封装在IP数据报内，以IP数据报的形式进行传输的。关于ICMP的正式规范详情参见RFC 792。ICMP报文的格式如图5-3所示，所有报文的前4个字节都是一样的，但是剩下的其他字节则互不相同。

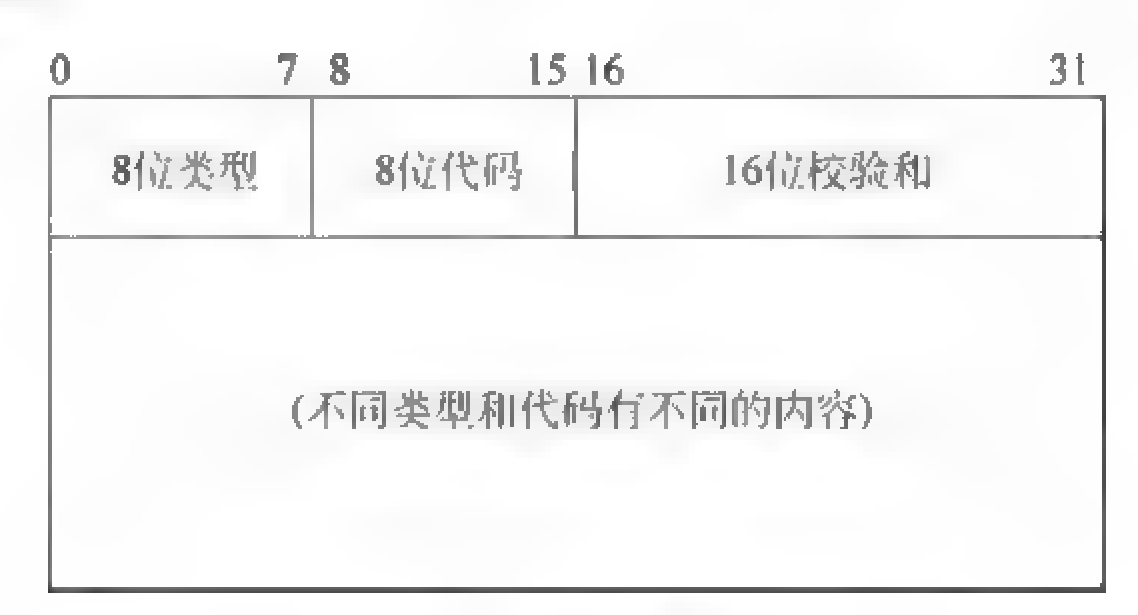


图5-3 ICMP报文格式

类型字段可以有15个不同的值，以描述特定类型的ICMP报文。某些ICMP报文还使用代码字段的值进一步描述引起差错的不同原因。校验和字段覆盖整个ICMP报文。

本章将选出一部分类型的ICMP报文进行详细介绍，包括地址掩码请求和应答、时间戳请求和应答以及不可达端口。

2. ICMP报文类型

在ICMP中定义了多种类型的ICMP报文，各种类型的ICMP报文如表5-1所示，报文类型由报文中的类型字段和代码字段共同决定。

表5-1中的最后两列表明了特定的ICMP报文是属于查询报文还是属于差错报文。因为有时需要对ICMP差错报文做特殊处理，因此需要进行区分。例如，在对ICMP差错报文进行响应时，永远不会生成另一份ICMP差错报文(如果没有这个限制规则，可能会遇到一

个差错产生另一个差错的情况，而差错再产生差错，这样会无休止地循环下去)。

表 5-1　ICMP 报文类型

类型	代码	描　述	查询	差错
0	0	回显应答(Ping 应答，第 7 章)	·	
3		目标不可达：		
	0	网络不可达(9.3 节)		·
	1	主机不可达(9.3 节)		·
	2	协议不可达		·
	3	端口不可达(6.5 节)		·
	4	需要进行分片但设置了不分片比特(11.6 节)		·
	5	源站选路失败(8.5 节)		·
	6	目标网络不认识		·
	7	目标主机不认识		·
	8	源主机被隔离(作废不用)		·
	9	目标网络被强制禁止		·
	10	目标主机被强制禁止		·
	11	由于服务类型 TOS，网络不可达(9.3 节)		·
	12	由于服务类型 TOS，主机不可达(9.3 节)		·
	13	由于过滤，通信被强制禁止		·
	14	主机越权		·
	15	优先权中止生效		·
4	0	源端被关闭(基本流控制，11.11 节)		·
5		复位向(9.5 节)：		·
	0	对网络复位向		·
	1	对主机复位向		·
	2	对服务类型和网络复位向		·
	3	对服务类型和主机复位向		·
8	0	请求回显(Ping 请求，第 7 章)	·	
9	0	路由器通知(9.6 节)	·	
10	0	路由器请求(9.6 节)	·	
11		超时：		
	0	传输期间生存时间为 0(Traceroute，第 8 章)		·
	1	在数据报组装期间生存时间为 0(11.5 节)		·
12		参数问题：		
	0	坏的 IP 首部(包括各种差错)		·
	1	缺少必需的选项		·
13	0	时间戳请求(6.4 节)	·	
14	0	时间戳应答(6.4 节)	·	
15	0	信息请求(作废不用)	·	
16	0	信息应答(作废不用)	·	
17	0	地址掩码请求(6.3 节)	·	
18	0	地址掩码应答(6.3 节)	·	

当发送一份ICMP差错报文时，报文始终包含IP的头部和产生ICMP差错报文的IP数据报的前8个字节。这样，接收ICMP差错报文就会把它与某个特定的协议(根据IP数据报头部中的协议字段判断)和用户进程(根据包含在IP数据报前8个字节中的TCP或UDP报文头部中的TCP或UDP端口号判断)联系起来。

需要说明的是，以下各种情况都不会导致产生ICMP差错报文。

(1) ICMP差错报文(但ICMP查询报文可能会产生ICMP差错报文)。

(2) 目标地址是广播地址或多播地址的IP数据报。

(3) 作为链路层广播的数据报。

(4) 不是IP分片的第一片。

(5) 源地址不是单个主机的数据报。这就是说，源地址不能为零地址、环回地址、广播地址或多播地址。

这些规则是为了防止ICMP差错报文对广播分组响应所带来的广播风暴。

3. ICMP地址掩码请求和应答

ICMP地址掩码请求用于无盘系统在引导过程中获取自己的子网掩码。系统广播它的ICMP请求报文(这一过程与无盘系统在引导过程中用RARP获取IP地址类似)。无盘系统获取子网掩码的另一个方法是BOOTP。ICMP地址掩码请求和应答报文的格式如图5-4所示。



图5-4 ICMP地址掩码请求与应答报文

ICMP报文中的标识符和序列号字段由发送端任意选择设定，这些值在应答中将被返回。这样，发送端就可以把应答与请求进行匹配。

4. ICMP时间戳请求与应答

ICMP时间戳请求允许系统向另一个系统查询当前的时间。返回的建议值是自午夜开始计算的毫秒数，被称为协调的统一时间(Coordinated Universal Time，UTC)。这种ICMP报文的好处是它提供了毫秒级分辨率，而利用其他方法从别的主机获取的时间(如某些UNIX系统提供的rdate命令)只能提供秒级分辨率。由于返回的时间是从午夜开始计算的，因此调用者必须通过其他方法获知当时的日期，这是它的一个缺陷。ICMP时间戳请求和应答报文格式如图5-5所示。

请求端填写发起时间戳，然后发送报文。应答系统收到请求报文时间填写接收时间戳，在发送应答时填写发送时间戳。但是实际上，大多数的实现把后面两个字段都设成相同的值。



图 5-5　ICMP 时间戳请求和应答报文

除此之外，还有其他方法可以获取时间和日期，包含：

(1) 日期服务程序和时间服务程序。前者是以人们可读的格式返回当前的时间和日期，是一行 ASCII 字符。可以用 Telnet 命令验证这个服务，时间服务程序返回的是一个 32 位的二进制数值，表示 UTC，1900 年 1 月 1 日午夜起算的秒数。这个程序是以秒为单位提供的日期和时间。

(2) 严格的计时器使用网络时间协议(NTP)，该协议在 RFC 1305 中给出了描述。这个协议采用先进的技术保证 LAN 或 WAN 上的一组系统时钟误差在毫秒级以内。

(3) 开放软件基金会(OSF)的分布式计算环境(DCE)定义了分布式时间服务(DTS)，它也提供计算机之间的时钟同步。

(4) 伯克利大学的 UNIX 系统提供守护程序(timed(8))同步局域网上的系统时钟。不像 NTP 和 DTS，timed 不在广域网范围内工作。

5. ICMP 端口不可达差错

端口不可达报文是 ICMP 目的不可到达报文中的一种，以此查看 ICMP 差错报文中所附加的信息。可使用 UDP 查看它。UDP 的规则之一是，如果收到一份 UDP 数据报而目标端口与某个正在使用的进程不相符，那么 UDP 返回一个不可达报文。可以用 TFTP 强制生成一个端口不可达报文。ICMP 不可达报文格式如图 5-6 所示。

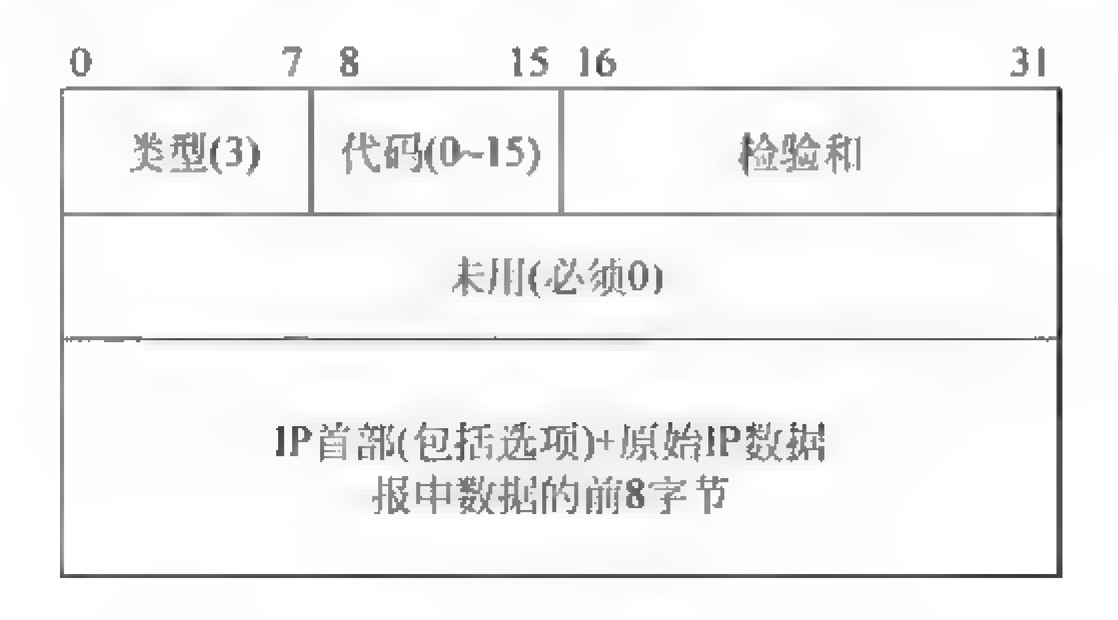


图 5-6　ICMP 不可达报文

5.2.2　ICMP 扫描过程

ICMP 扫描是基本的安全扫描技术，是很多其他扫描功能的基础，可以利用它判断对方

主机或网络设备是否在正常运行。只有确定其正常运行，才能够继续执行后续的各种扫描功能。但是，有时候使用ICMP扫描不一定准确，因为很多系统都安装了各种安全软件应对ICMP扫描，另外，用户也可以通过防火墙或路由器的相关设置以防止ICMP扫描。ping程序就是利用ICMP实现的，主要是利用ICMP最基本的报错功能，根据网络协议，如果出现了错误，接收端将产生一个ICMP的错误报文。这些错误报文并不是主动发送的，而是由错误触发，根据特定的协议格式自动产生的。

当IP数据报出现校验和与版本错误时，目标主机将抛弃这个数据报，如果校验和出现错误，则路由器直接丢弃该数据报。有些主机比如AIX、HP-UX等，是不会发送ICMP的不可达数据报的。

ICMP报文被封装成IP包时，是作为IP层数据包的数据部分，在前面加上数据包的首部，组成IP数据包发送出去。最常用的ICMP扫描方法就是利用ping的原理，发送ICMP回显请求报文，然后监听是否有ICMP回显应答报文返回，如果没有就证明目标主机不存在或者已经停机，如果能够返回ICMP回显应答报文，就证明主机正在运行。其过程如图5-7所示。

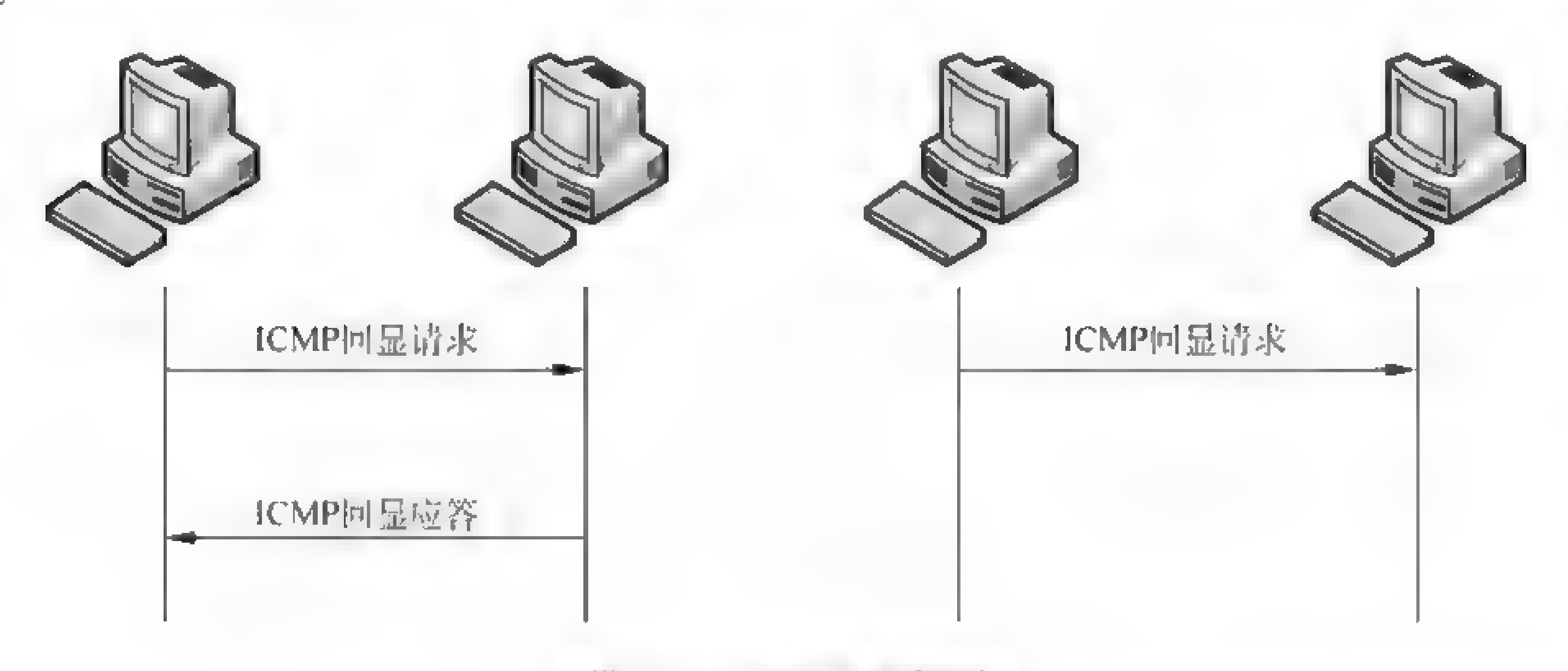


图5-7 ICMP回显探测

也可以利用异常IP包进行探测，例如，向目标主机发送错误的IP包，目标主机会反馈ICMP Parameter Problem Error信息。常见的伪错误字段为Header Length和IP Options。不同厂家的路由器和操作系统对这些错误的处理方式不同，返回的结果也不同。这种方法同样可以对目标主机和网络设备进行探测。

另外，可以构造超长包探测路由器，若构造的数据包长度超过目标路径上路由器的路径最大传输单元(PMTU)，且在数据包内设置禁止分片标志，那么该路由器会反馈一个差错报文，其内容表示当前数据包要通过该路径需要分片，但是设置了不分片标志位DF(Don't Fragment)，因此无法通过。

如果目标主机是在防火墙内部，可以利用反向探测技术探测被过滤设备或防火墙保护的网络和主机。构造可能的内部IP地址列表，并向这些地址发送数据包。当对方路由器接收到这些数据包时，会进行IP识别并路由，对不在其服务范围的IP包发送ICMP Host Unreachable或ICMP Time Exceeded错误报文，没有接到相应错误报文的IP地址可被认为在该网络中。

5.3　TCP 扫描

利用 TCP 进行端口扫描是常用的端口扫描方法，因为现在的很多网络应用都是基于 TCP 实现的，例如，Web 服务器就是基于 TCP 端口 80 的。最基本的 TCP 扫描就是使用 TCP 连接实现，如果目标主机能够连接成功，就表示对方开放了此端口；如果失败就表示端口关闭。在 TCP 连接扫描过程中实现了正常的 TCP 的连接过程，但是这种方式需要的时间比较长，而且容易产生审计数据，所以有很多限制。

5.3.1　TCP 协议

TCP 是一个面向连接的协议，首先了解 TCP 的数据格式，如图 5-8 所示。

32b

TCP头部

<table>
<tr><td colspan="8">源端口</td><td>目标端口</td></tr>
<tr><td colspan="9">序号</td></tr>
<tr><td colspan="9">确认序号</td></tr>
<tr><td>数据偏移</td><td>保留</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td>窗口</td></tr>
<tr><td colspan="8">检验和</td><td>紧急指针</td></tr>
<tr><td colspan="9">选项和填充</td></tr>
<tr><td colspan="9">数据</td></tr>
</table>

图 5-8　TCP 报文头部

Source Port 是源端口，16 位。

Destination Port 是目标端口，16 位。

Sequence Number 是发送数据包中的第一个字节的序列号，32 位。

Acknowledgment Number 是确认序列号，32 位。

Data Offset 是数据偏移，4 位，该字段的值是 TCP 头部(包括选项)长度除以 4。

标志位：6 位，URG 表示 Urgent Pointer 字段有意义；ACK 表示 Acknowledgment Number 字段有意义；PSH 表示 Push 功能，RST 表示复位 TCP 连接；SYN 表示 SYN 报文(在建立 TCP 连接的时候使用)；FIN 表示没有数据需要发送了(在关闭 TCP 连接的时候使用)；Window 表示接收缓冲区的空闲空间，16 位，用于告诉 TCP 连接对端自己能够接收的最大数据长度。

Checksum 是校验和，16 位。

Urgent Pointers 是紧急指针，16 位，只有 URG 标志位被设置时该字段才有意义，表示

紧急数据相对序列号(Sequence Number 字段的值)的偏移。

5.3.2 TCP 扫描过程

TCP 正常连接的建立和中止过程如图 5-9 所示。

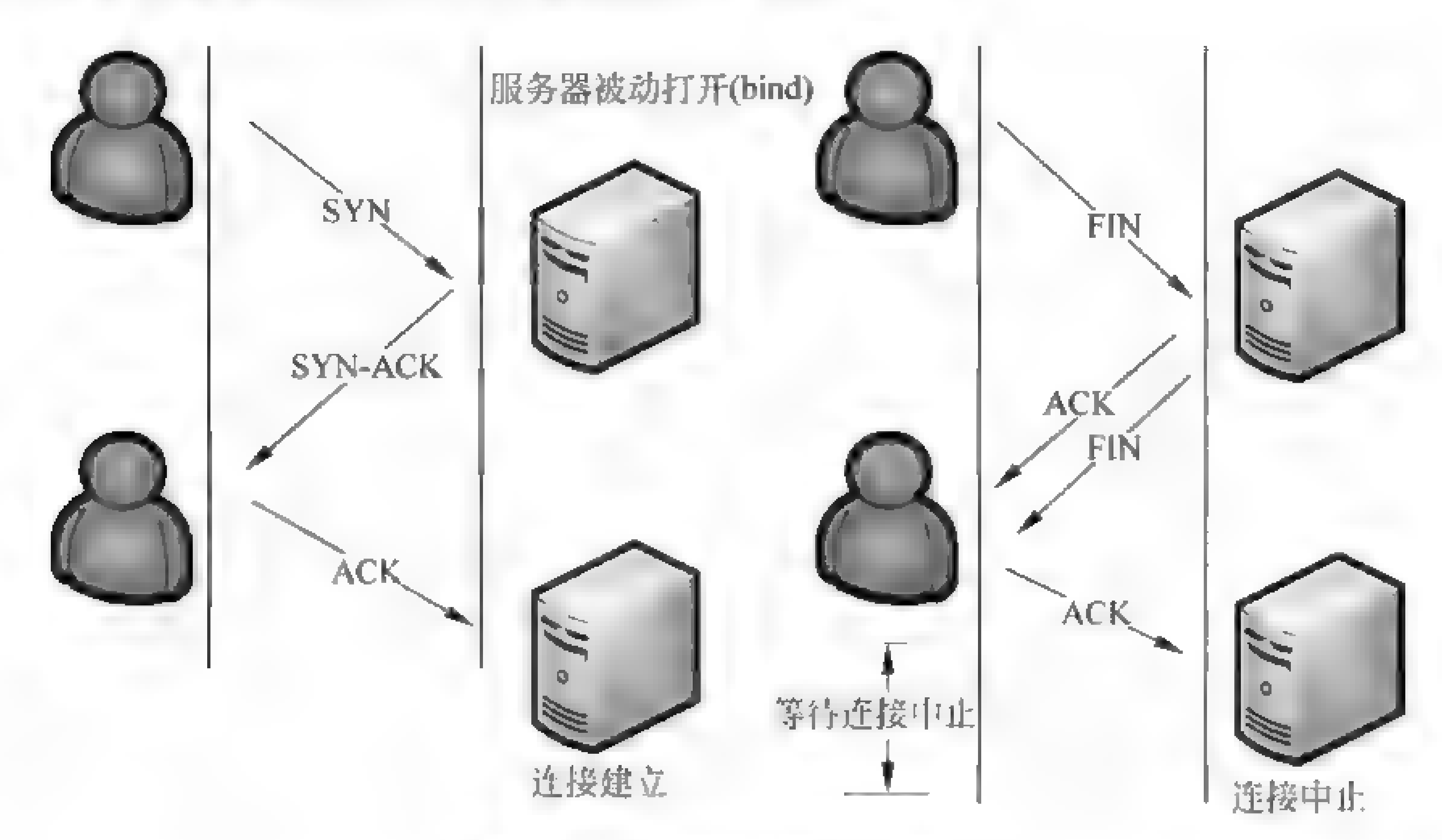


图 5-9 TCP 连接的建立和终止

TCP 用 3 个报文建立一个连接，而终止连接时则需要 4 个报文，原因是被动关闭连接方需要关闭处理时间，因此，ACK 和 FIN 不能同时发给主动关闭方。

在高级的 TCP 扫描技术中主要利用 TCP 连接的 3 次握手特性进行，也就是所谓的半开扫描。这些办法可以绕过一些防火墙，在不被欺骗的情况下得到防火墙后面的主机信息。这些方法还有一个好处是难以被记录，不容易被系统发现。

基于 TCP 连接的扫描过程如图 5-10 所示，根据 TCP/IP，函数 connect()会激发 TCP 的 3 次握手过程。如果使用 Socket 函数 connect()连接成功，表示端口是开放的，如果返回错误，则表示端口关闭。

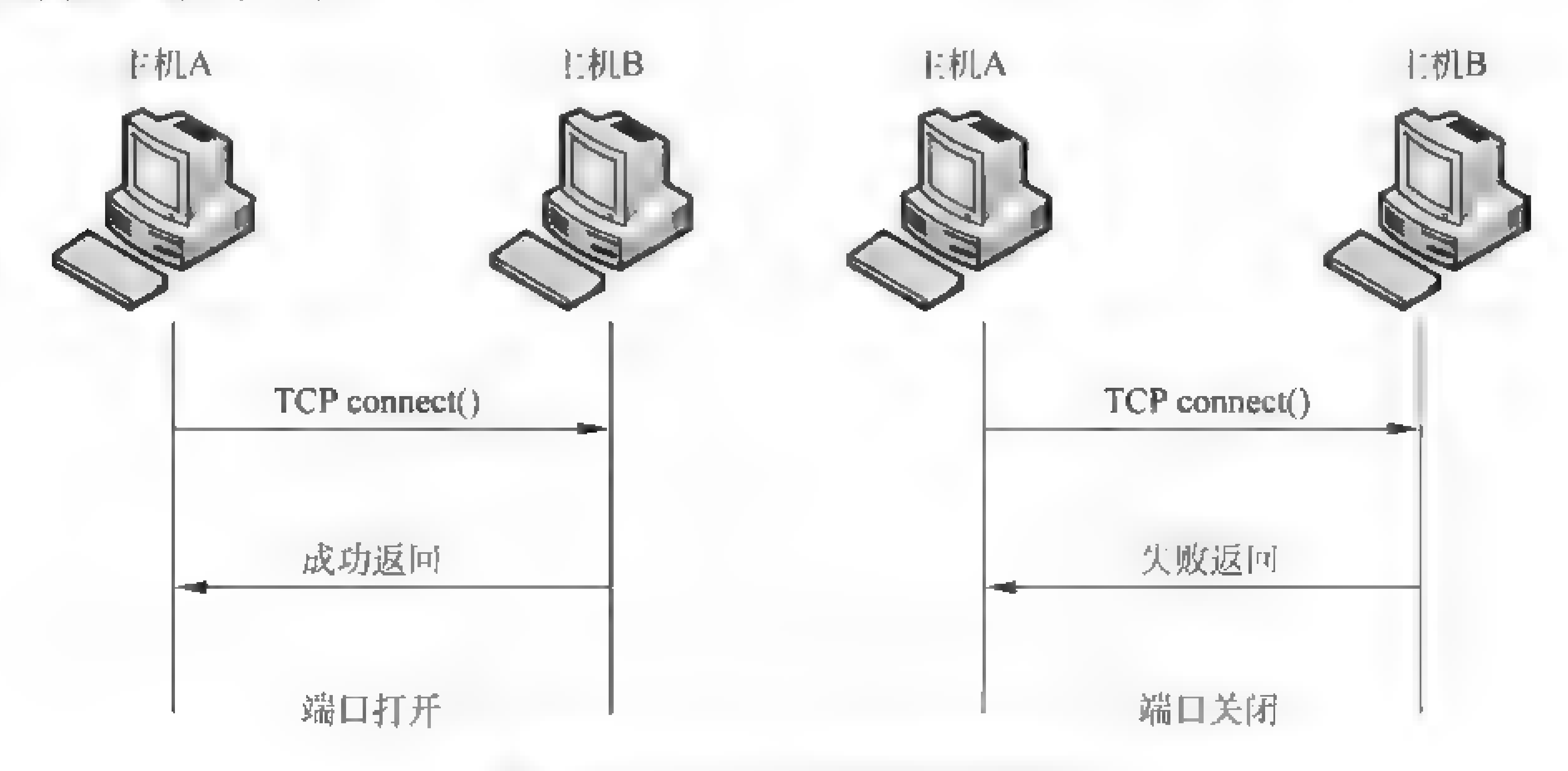


图 5-10 基于 TCP 连接的扫描过程

5.3.3　TCP 扫描分类

1. TCP connect()扫描

这是最基本的 TCP 扫描，由操作系统提供的 connect()系统调用是用来与每个感兴趣的目标计算机的端口进行连接。如果端口处于侦听状态，connect()就能成功，否则，这个端口是不能用的，即没有提供服务。该技术最大的优点之一是系统用户不需要任何权限，任何用户都有权利使用这个调用。另一个好处就是速度快，如果对每个目标端口以线性的方式使用单独的 connect()调用，将会花费相当长的时间，因此用户可以通过同时打开多个套接字进行加速扫描。使用非阻塞 I/O 允许用户设置一个较低的时间用尽周期，同时可观察多个套接字，但这种方法的缺点是很容易被发觉，并被过滤掉，因为目标计算机的 logs 文件中会显示一连串的连接和连接时出错的服务消息，并且能很快地使它关闭。

2. TCP SYN 扫描

在 TCP SYN 扫描中，扫描程序不需要打开一个完全的 TCP 连接，因此该扫描通常被认为是"半开放"扫描。如图 5-11 所示，扫描程序发出了一个 SYN 数据包，看起来好像准备建立一个实际的连接并等待对方反应(参考 TCP 三次握手法建立连接的过程)，如果返回了一个 SYN|ACK 的信息，表示目标端口处于侦听状态；返回一个 RST 信息，表示端口没有处于侦听状态。如果收到一个 SYN|ACK，则扫描程序必须再发送一个 RST 信号关闭该连接。TCP SYN 扫描技术一般不会在目标计算机上留下记录，但这种方法要求扫描者必须要有 root 权限才能建立自己的 SYN 数据包。

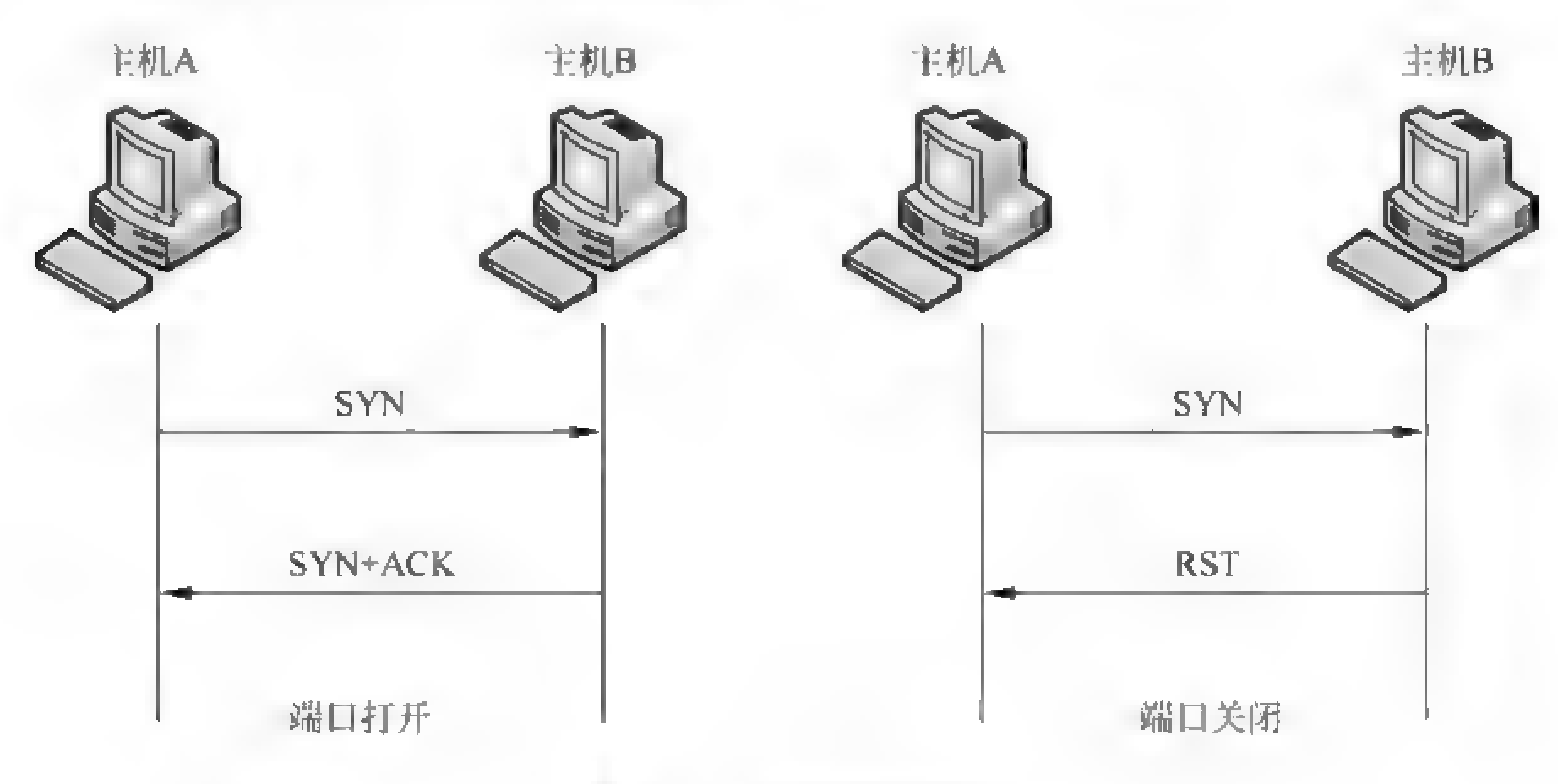


图 5-11　TCP SYN 扫描

3. TCP FIN 扫描

有的时候 SYN 扫描可能都不够秘密，因为一些防火墙和包过滤器会对某些指定的端口进行监视，部分程序能检测到这些扫描行为。相反，FIN 数据包有可能在不引起任何麻烦的情况下顺利通过。这种扫描方法的出发点是关闭的端口会用适当的 RST 回复 FIN 数据包，而打开的端口会忽略对 FIN 数据包的回复。这种方法受系统的实现方式所影响，如果系统不管端口是否打开都一律回复 RST 的话，该扫描方法就不适用了。并且这种方法在区分 UNIX 和 NT 时，是十分有用的。

如图5-12所示，构造一个含有FIN标志的TCP数据包，将其送到目标主机B的某一个端口，如果返回含有RST的TCP报文，表示端口关闭；如果没有任何反应，有可能表示端口打开；如果产生ICMP差错报文，则端口的状态是未知。

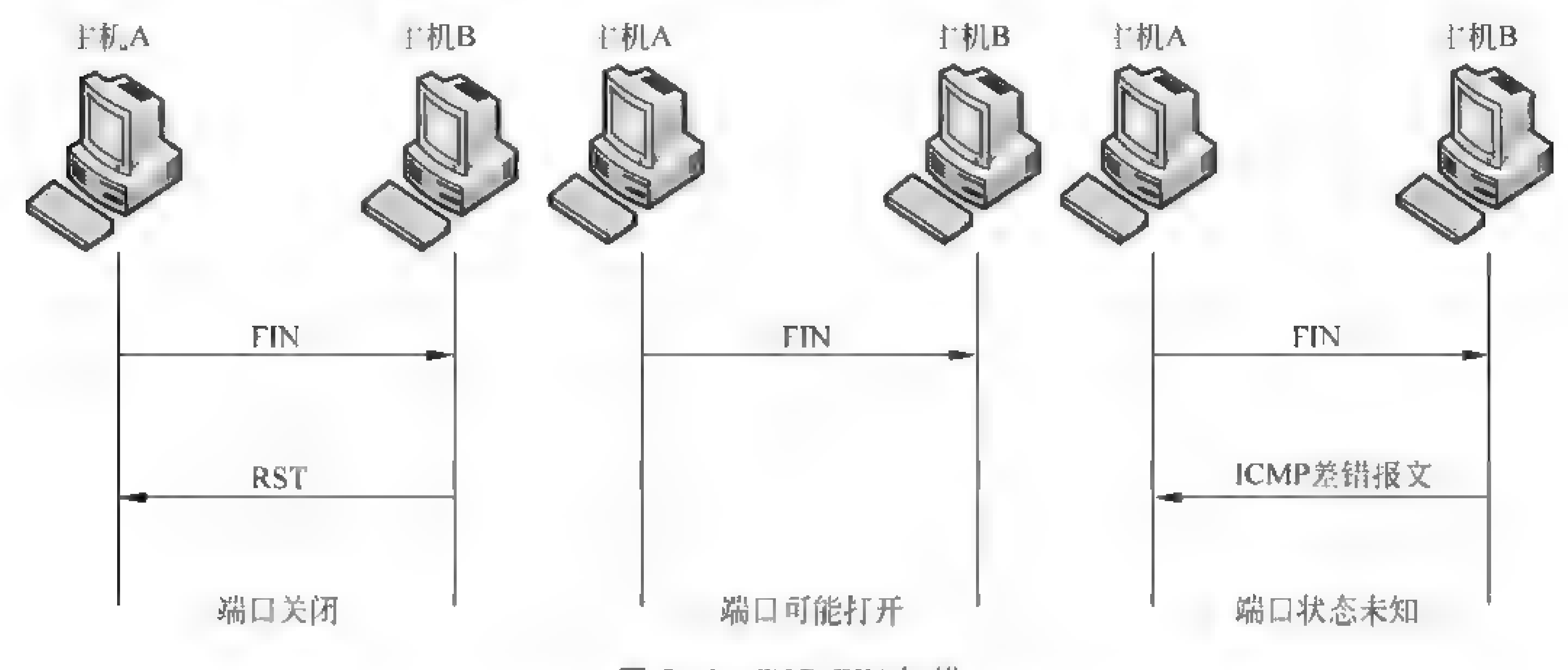


图5-12 TCP FIN扫描

上文的"有可能"是指由于网络环境的复杂性，或者由于有防火墙或存在其他网络过滤设备，阻碍了正常的数据流程，所以不能判断端口是否打开。需要注意的是，对Windows系统，TCP FIN扫描也是无效的。

4. IP段扫描

该方法不算是新方法，只是其他技术的变化。它并不是直接发送TCP探测数据包，而是将数据包分成两个较小的IP段。这样就将一个TCP头分成几个数据包，这样过滤器就很难探测到该扫描。但必须小心，一些程序在处理这些小数据包时会有些麻烦。

5. TCP反向ident扫描

通过ident协议(RFC 1413)，可以发现基于TCP连接的任何进程的用户名，即使这个连接不是由此进程开始的。因此，用户能连接到http端口，然后用ident发现服务器是否正在以root权限运行。当然，只有在与目标端口建立了一个完整的TCP连接后，使用此方法才能达到效果。

6. FTP返回攻击

FTP协议有一个有趣的特点，它支持代理(Proxy)FTP连接，即入侵者可以从自己的计算机向目标主机的FTP server PI(协议解释器)发起连接，建立一个控制通信连接，然后，请求这个server PI激活一个有效的server DTP(数据传输进程)给Internet上任何地方发送文件。

7. ACK扫描

TCP ACK扫描是利用ACK标识位进行的攻击方式，ACK标识在TCP中表示确认序号有效，它表示确认一个正常的TCP连接。但是在TCP ACK扫描中没有进行正常的TCP连接过程，那么当发送一个带有ACK标识的TCP报文到目标主机端口时，目标主机会怎样反应呢？

当给目标主机发送一个含有 ACK 标识的 TCP 报文的时候，无论目标端口是开放或者关闭，对方都会返回一个含有 RST 标志的报文。因此，使用 TCP ACK 扫描并不能确定目标端口的状态为关闭还是开放。但仍旧可利用该方法扫描防火墙的配置，发现防火墙的规则，确定它们是有状态的还是无状态的，哪些端口是被过滤的。

8. NULL 扫描

TCP NULL 扫描的原理是将不设置任何标识位的 TCP 数据包发送给目标主机，如果目标主机的相应端口是关闭的，应该返回一个 RST 数据包，如果目标主机端口是打开的，则没有任何反应，它的表现与 TCP FIN 扫描是一样的。同样需要注意的是，对 Windows 系统，NULL 扫描也是无效的。

5.4　UDP 扫描

与 TCP 类似，UDP 也是使用端口号进行数据传输的，因此，如何识别 UDP 端口的状态，是必须考虑的问题。一般情况下，当向一个关闭的 UDP 端口发送数据时，目标主机会返回一个 ICMP 不可达的错误。UDP 扫描就是利用了上述原理，首先构造一个空 UDP 报文，将其发送给目标主机的特定端口，如果返回消息显示为 ICMP 端口不可达错误，表示该端口是关闭的；如果是其他的 ICMP 不可达差错报文，表示端口状态是未知的；如果返回一个 UDP 报文，表示该端口是开放的；如果没有任何报文响应，则表示该端口有可能是开放的。

在 UDP 扫描中，多是与 ICMP 组合进行的。还有一些特殊的就是 UDP 回馈，比如 SQL Server，对其 1434 端口发送“x02”就能够探测到其连接端口。UDP 扫描过程如图 5-13 所示。

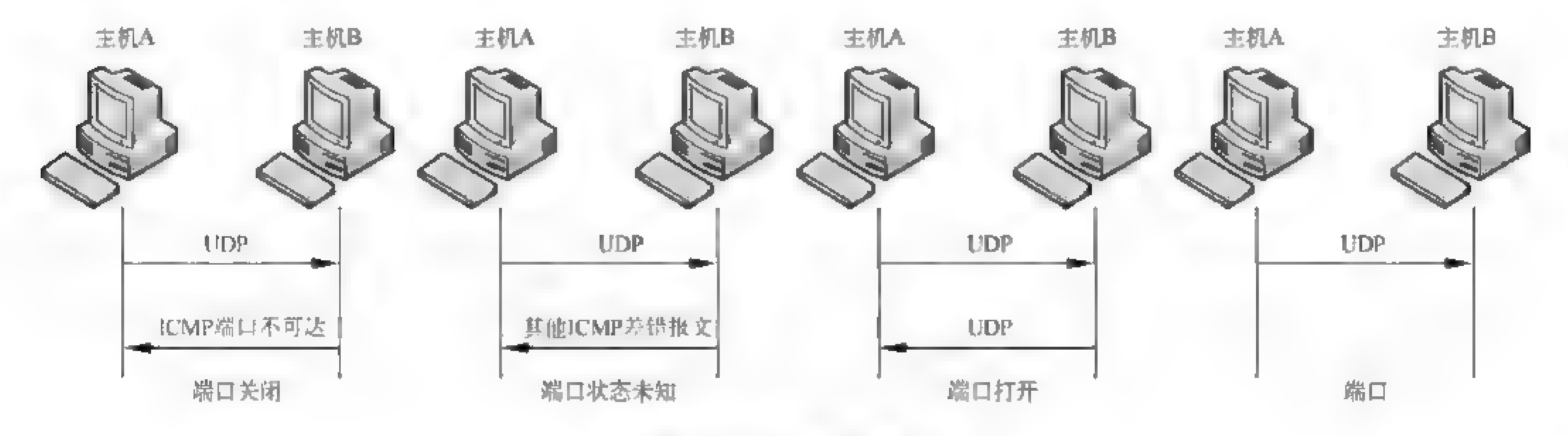


图 5-13　UDP 扫描

在图 5-13 中，主机 A 向主机 B 的某端口发送 UDP 数据包，如果返回一个 UDP 数据包，表示该端口是开放的；如果返回 ICMP 端口不可达报文，表示该端口是关闭的；如果没有任何数据返回，该端口有可能是开放的；如果返回其他的 ICMP 差错报文，则端口状态未知。

除此之外，扫描手段还可按照不同的方式进行分类，上述的 TCP 和 UDP 扫描方式可以被划分为以下类型。

1. 开放扫描

TCP 连接扫描属于开放扫描，需要扫描方通过 3 次握手过程与目标主机建立完整的

TCP 连接，可靠性高，但是在扫描的过程中会产生大量的审计数据，这种扫描在某些场合非常有用，它可以作为正常扫描行为，直接由网络管理员使用。

2. 半开放扫描

开放扫描需要 TCP 进行一个正常的连接，而半开放扫描不需要正常连接，只是一个部分行为。扫描的时候只是利用 TCP 正常连接的某一个部分完成操作。常见的半开放扫描方法有 TCP SYN 扫描、TCP ACK 扫描等。

3. 隐秘扫描

在半开放扫描中还包括 TCP 连接的某一部分行为，但是在隐秘扫描中，则不包括任何 TCP 三次握手协议的任何部分，其隐蔽性好，但这种扫描使用的数据包在通过网络时容易被丢弃从而产生错误的探测信息，效果不是很好。这种扫描方法包括：TCP FIN 扫描、TCP XMAS 扫描、NULL 扫描和 UDP 扫描等。

5.5 木马扫描

木马是一种经过伪装的欺骗性程序。木马程序与一般的病毒不同，它不会自我繁殖，也并不能感染其他文件，它的主要作用是为施木马者打开种有木马计算机的门户，使其可以任意毁坏、窃取目标主机的文件，甚至远程操控目标主机。

一个典型的木马程序包含两部分：客户端和服务器。把服务器程序放入远程目标主机中，客户端则由攻击者进行掌控，攻击者利用客户端连接服务器，远程控制目标主机。虽然目前出现了很多新的木马形式，但其原理大同小异，还有一些为完成特定任务设计的木马。

既然木马设计是基于服务器和客户端模式，那么它一定会有一个开放的端口监听连接，当然，不同的木马使用不同的端口号，可以通过扫描特定的木马端口发现主机是否被某木马入侵。但是，某些新型的木马并不开放特定端口，而是捆绑到合法的应用程序中，这让用户很难扫描发现。

5.6 漏洞扫描

5.6.1 漏洞扫描技术

漏洞扫描技术是一类重要的网络安全技术，它与防火墙、入侵检测系统互相配合，能够有效提高网络的安全性。通过对网络的扫描，网络管理员能了解网络的安全设置和运行的应用服务，及时发现安全漏洞，客观评估网络风险等级。除此之外，网络管理员还能根据扫描结果更正网络安全漏洞和系统中的错误设置，在黑客攻击前及时进行防范。如果说防火墙和网络监视系统是被动的防御手段，那么安全扫描就是一种主动的防范措施，能有效避免黑客攻击行为，做到防患于未然。

总的来说，漏洞扫描的目的主要包含以下几个方面。

1. 定期的网络安全自我检测、评估

配备漏洞扫描系统，网络管理人员可以定期进行网络安全检测服务，而安全检测可帮助客户最大可能地消除安全隐患，尽可能早地发现安全漏洞并进行修补，有效地利用已有系统，优化资源，提高网络的运行效率。

2. 安装新软件、启动新服务后的检查

由于漏洞和安全隐患的形式多种多样，安装新软件和启动新服务都有可能使原来隐藏的漏洞暴露出来，因此进行这些操作之后应该重新扫描系统，从而使安全得到保障。

3. 网络建设和网络改造前后的安全规划评估和成效检验

网络建设者必须建立整体安全规划，以统领全局，高屋建瓴。在可以容忍的风险级别和可以接受的成本之间，取得恰当的平衡，在多种多样的安全产品和技术之间做出取舍。配备网络漏洞扫描/网络评估系统可以让用户很方便地进行安全规划评估和成效检验。

4. 网络承担重要任务前的安全性测试

网络承担重要任务前应该多采取主动安全措施，防止出现事故，从技术上和管理上加强对网络安全和信息安全的重视，形成立体防护，由被动修补变成主动防范，最终把事故的概率降到最低。配备网络漏洞扫描/网络评估系统可以让用户很方便地进行安全性测试。

5. 网络安全事故后的分析调查

网络安全事故后可以通过网络漏洞扫描/网络评估系统来分析确定网络被攻击的漏洞所在，帮助弥补漏洞，尽可能多地提供资料，方便调查攻击的来源。

6. 重大网络安全事件前的准备

在发生重大网络安全事件之前，网络漏洞扫描/网络评估系统能够帮助用户及时找出网络中存在的隐患和漏洞，并及时进行弥补。

7. 公安、保密部门组织的安全性检查

互联网安全主要分为网络运行安全和信息安全两部分。网络运行安全主要包括以ChinaNet、ChinaGBN、CNCnet 等十大计算机信息系统的运行安全和其他专网的运行安全；信息安全包括接入 Internet 的计算机、服务器、工作站等用来进行采集、加工、存储、传输、检索处理的人机系统的安全。网络漏洞扫描/网络评估系统能够积极地配合公安、保密部门组织进行安全性检查。

5.6.2 漏洞扫描分类及技术

依据扫描执行方式不同，漏洞扫描主要分为以下 3 类。

1. 针对网络的扫描器

基于网络的扫描器就是通过网络扫描远程计算机中的漏洞。

2. 针对主机的扫描器

基于主机的扫描器则是在目标系统上安装了一个代理（Agent）或者是服务（Services），以便能够访问所有的文件与进程，这也使得基于主机的扫描器能够扫描到更多的漏洞。二者相比，基于网络的漏洞扫描器的价格相对比较便宜，在操作过程中，不需要涉及目标系统

的管理员，在检测过程中不需要在目标系统上安装任何东西，维护简便。

3. 针对数据库的扫描器

主流数据库的自身漏洞逐步暴露，数量庞大，仅 CVE 公布的 Oracle 漏洞数已达一千一百多个。数据库漏洞扫描可以检测出数据库的 DBMS 漏洞、默认配置、权限提升漏洞、缓冲区溢出、补丁未升级等自身漏洞。

除了上述的漏洞扫描技术之外，还有针对 Web 应用、中间件等其他扫描技术，此处不再一一列举。

具体的扫描技术包含以下 7 种。

(1) 主机扫描：确定在目标网络上的主机是否在线。

(2) 端口扫描：发现远程主机开放的端口以及服务。

(3) OS 识别技术：根据信息和协议栈判别操作系统。

(4) 漏洞检测数据采集技术：按照网络、系统、数据库进行扫描。

(5) 智能端口识别、多重服务检测、安全优化扫描、系统渗透扫描。

(6) 多种数据库自动化检查技术，数据库实例发现技术。

(7) 多种 DBMS 的密码生成技术，提供口令爆破库，实现快速的弱口令检测方法。

5.7 实例编程——端口扫描实现

5.7.1 ICMP 扫描实现

1. ICMP 扫描实现流程

ICMP 扫描是利用 ICMP 实现的，在本例中使用了基本的 ICMP Echo 查询报文和 Echo 应答报文实现扫描，具体流程如图 5-14 所示。

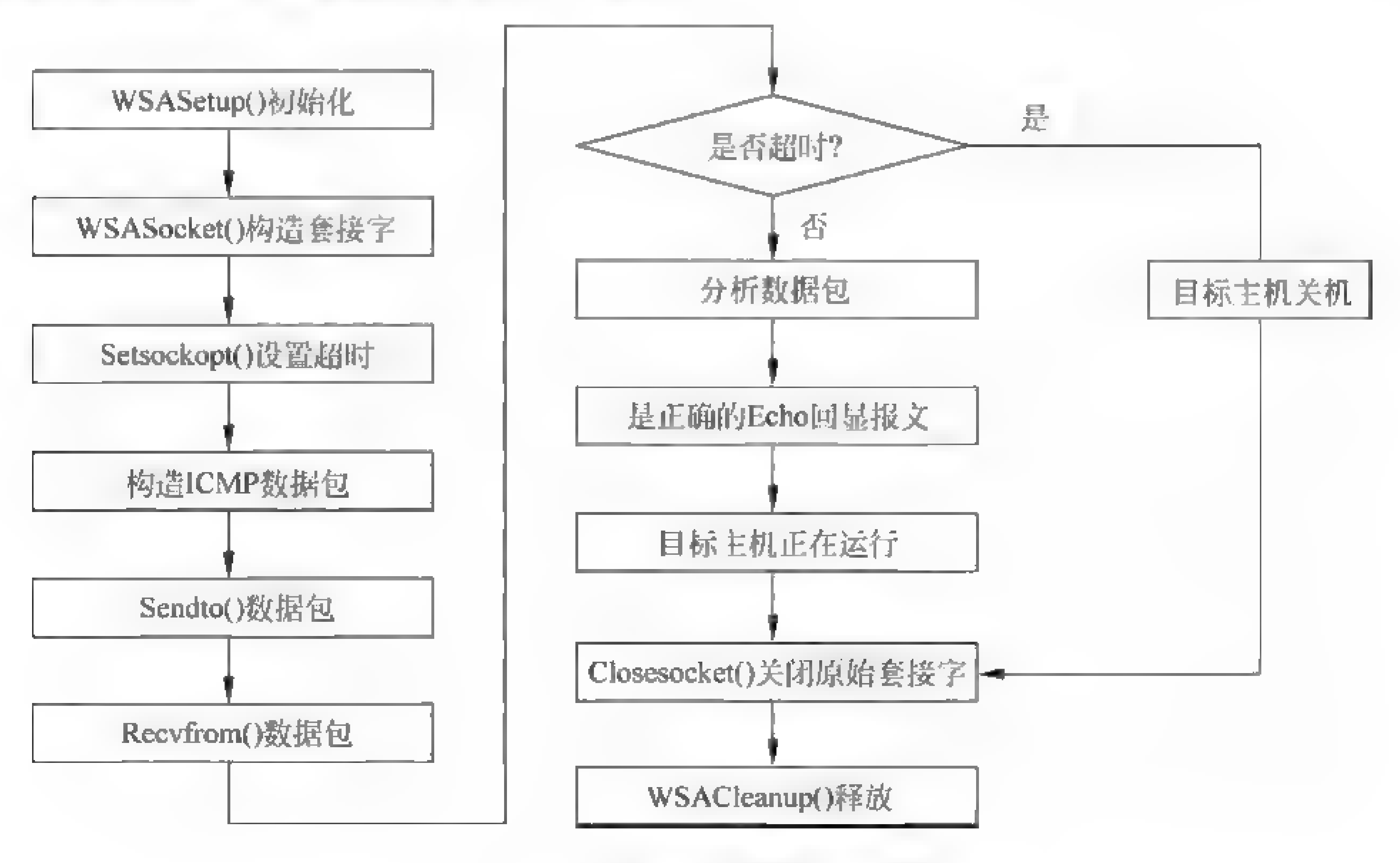


图 5-14 ICMP 扫描实现过程

2. ICMP 扫描实现代码

ICMP 实现代码如下。

```
(运行此程序的时候,要以管理员身份运行)
//
#include "stdafx.h"
#include "stdio.h"
#include "Winsock.h"
#pragma comment(lib,"ws2_32.lib")
typedef struct IpHeader
{
unsigned char Version_HLen;            //头部长度和 IP 版本号
unsigned char TOS;                     //服务器类型 TOS
unsigned short Length;                 //总长度
unsigned short Ident;                  //标识
unsigned short Flags_Offset;           //标志位
unsigned char TTL;                     //生存时间 TTL
unsigned char Protocol;                //协议(TCP 或其他)
unsigned short Checksum;               //IP 头部校验和
unsigned int SourceAddr;               //源 IP 地址
unsigned int DestinationAddr;          //目标 IP 地址

}Ip_Header;
typedef struct IcmpHeader
{
    BYTE Type;                         //8 位类型
    BYTE Code;                         //8 位代码
    USHORT Checksum;                   //16 位校验和
    USHORT ID;                         //16 位识别号
    USHORT Sequence;                   //16 位报文序列号
}Icmp_Header;
//计算机校验和
USHORT checksum(USHORT * buff, int size)
{
    unsigned long cksum = 0;
    while (size > 1)
    {
        cksum += *buff++;
        size -= sizeof(USHORT);
    }
    if (size)
    {
        cksum += *(UCHAR *)(buff);
    }
    cksum = (cksum >> 16);
    return(USHORT)(~cksum);
}
//主函数
int main(int argc, char * argv[])
```

```
{
    //WSADATA 数据结构
    WSADATA wsaData;
    //目标地址结构
    sockaddr_in DestAddr;
    //IP 头部
    Ip_Header * ip;
    //icmp 头部
    Icmp_Header * icmp;
    //发送 ICMP 的数据
    Icmp_Header * SendIcmp;
    //设置超时
    int Timeout = 1000;
    //目标 IP 地址
    char DestIpAddr[100] = "10.171.68.54";
    //构造 ICMP 数据内容
    char IcmpBuffer[8] = "";
    //ICMP 套接字
    SOCKET IcmpSocket;
    //永远接收数据
    char RecvBuffer[1024];
    //地址结构
    sockaddr_in addr;
    int Len = sizeof(addr);
    //返回值
    int Result;
    //初始化 Winsock
    if ((Result = WSAStartup(MAKEWORD(2, 2), &wsaData)) != 0)
    {
        printf("WSAStartup failed with error % d\n", Result);
        return 0;
    }
    IcmpSocket = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
    if (IcmpSocket == INVALID_SOCKET)
    {
        printf("socket failed with error % d\n", WSAGetLastError());
        return 0;
    }
    //设置套接字属性
    //设置超时特性
    Result = setsockopt(IcmpSocket, SOL_SOCKET, SO_RCVTIMEO, (char * )&Timeout, sizeof(Timeout));
    if (Result == SOCKET_ERROR)
    {
        printf("setsockopt failed with error % d\n", WSAGetLastError());
        return 0;
    }
    //填充目标地址结构
    memset(&DestAddr, 0, sizeof(DestAddr));
    DestAddr.sin_addr.s_addr = inet_addr(DestIpAddr);
    DestAddr.sin_port = htons(0);
```

```
    DestAddr.sin_family = AF_INET;
    //填充 ICMP 数据内容
    SendIcmp = (Icmp_Header * )IcmpBuffer;
    //填充类型
    SendIcmp->Type = 8;
    //填充代码
    SendIcmp->Code = 0;
    //填充 ID 号
    SendIcmp->ID = (USHORT)GetCurrentProcessId();
    //填充序列号
    SendIcmp->Sequence = htons(1);
    SendIcmp->Checksum = 0;
    //计算校验和
    SendIcmp->Checksum = checksum((USHORT * )IcmpBuffer, sizeof(IcmpBuffer));
    /* 发送数据 ICMP 数据 */
    Result = sendto(IcmpSocket, IcmpBuffer, sizeof(IcmpBuffer), 0, (SOCKADDR * )&DestAddr,
sizeof(DestAddr));
    if (Result == SOCKET_ERROR)
    {
        printf("sendto failed with error %d\n", WSAGetLastError());
        return 0;
    }
    Result = recvfrom(IcmpSocket, RecvBuffer, 1024, 0, (sockaddr * )&addr, &Len);
    if (Result == SOCKET_ERROR)
    {
        if (WSAGetLastError() != WSAETIMEDOUT)
        {
            printf("recvfrom failed with error %d\n", WSAGetLastError());
            return 0;
        }
        else
        {
            //超时
            printf("Host %s may be down.\n", DestIpAddr);
        }
    }
    if (Result < sizeof(Ip_Header) + sizeof(Icmp_Header))
    {
        printf("data error from %d\n", inet_ntoa(addr.sin_addr));
    }
    //读取返回的数据
    ip = (Ip_Header * )RecvBuffer;
    if ((ip->SourceAddr == inet_addr(DestIpAddr)) && (ip->Protocol == IPPROTO_IGMP))
    {
        //读取 ICMP 数据
        icmp = (Icmp_Header * )(RecvBuffer + sizeof(Ip_Header));
        //判断是否是应答
        if (icmp->Type != 0)
        {
            printf("type error %d", icmp->Type);
```

```
            return 0;
        }
        if (icmp->ID != GetCurrentProcessId())
        {
            printf("id error %d\n", icmp->ID);
            return 0;
        }
        else if ((icmp->Type == 0) && (icmp->ID == GetCurrentProcessId()))
        {
            printf("Host %s is up.\n", DestIpAddr);
        }
}
    //关闭套接字
    if (closesocket(IcmpSocket) == SOCKET_ERROR)
    {
        printf("clsoesocket failed with error %d\n", WSAGetLastError());
        return 0;
    }
    //释放 Winsock
    if (WSACleanup() == SOCKET_ERROR)
    {
        printf("WSACleanup failed with error %d\n", WSAGetLastError());
        return 0;
    }
    return 1;
}
```

运行结果如图 5 15 所示。



图 5-15 ICMP 扫描运行结果图

在本程序中，特别要注意设置超时这个步骤，因为如果目标主机不存在会关闭，就不可能得到响应，由于 recvfrom( )是阻塞性质的，它会一直等待数据的到达，这样程序就不会停止。超时设置代码如下：

```
//设置套接字属性
//设置超时属性
    Result = setsockopt (IcmpSocket, SOL_SOCKET, SO_RCVTIMEO,
    (char *)&Timeout , sizeof(Timeout));
   if (Result == SOCKET_ERROR){
      printf("setsockopt failed with error %d\n",WSAGetLastError( ));
      return 0;
   }
```

接收数据的时候，如果没有数据到达，那么会等待 1s 之后退出，进入超时状态，然后就可以判断对方主机是关闭的。本程序构造了 ICMP 回显请求数据包。

```
//填充 ICMP 数据内容
SendIcmp = (Icmp_Header * ) IcmpBuffer;
//填充类型
SendIcmp -> Type = 8;
//填充代码
SendIcmp -> Code = 0;
//填充 ID 号
SendIcmp -> ID  = (USHORT)GetCurrentProcessID();
//填充序列号
SendIcmp -> Sequence = htons(1);
SendIcmp -> Checksum = 0;
//计算校验和
SendIcmp -> Checksum = checksum(USHORT * )IcmpBuffer,sizeof(IcmpBuffer));
```

回显请求的类型为 8，代码值为 0，ID 号是用进程的 ID 填充的，然后计算机 ICMP 数据的校验和。

在接收数据包的时候，必须对数据包进行正确分析，发送回显请求数据包的目的是获得正确的回显应答数据包，如果不是回显应答就抛弃。

```
//读取返回的数据
ip = (Ip_Header  * )RecvBuffer;
      if ((ip -> SourceAddr == inet_addr(DestIpAddr))&&(ip -> Protocol == IPPROTO_ICMP))
      {
          //读取 ICMP 数据
           Icmp = (Icmp_Header  * ) (RecvBuffer + sizeof(Ip_Header));
        //判断是否是应答
          if (icmp -> Type!= 0){
           printf("type error  % d",icmp -> Type);
           return 0;
          }
If(icmp -> ID!= GetCurrentProcessId(   )){
      printf("id error  % d\n",icmp -> ID);
      Return 0;
   }
  else if((icmp -> Type == 0)&& (icmp -> ID == GetCurrentProcessId( )))
    {
     Printf("Host  % s is up.\n",DestIpAddr);
     }
}
```

回显请求的数据包内容是 ICMP 类型为 0，而且 ID 号也要一致，如果能够正确获得符合要求的回显应答，就判断目标主机是正在运行的。

5.7.2　TCP 扫描实现

1. TCP 连接扫描实现

1) TCP 连接扫描实现流程

TCP 连接扫描利用了 TCP 的正常连接过程，如果连接正常远程主机端口，表示端口是

开放的；如果连接失败，则表示端口关闭。

本程序实现利用TCP连接扫描方法对一个端口进行循环扫描，判断端口是否打开或者关闭，流程图如图5-16所示。

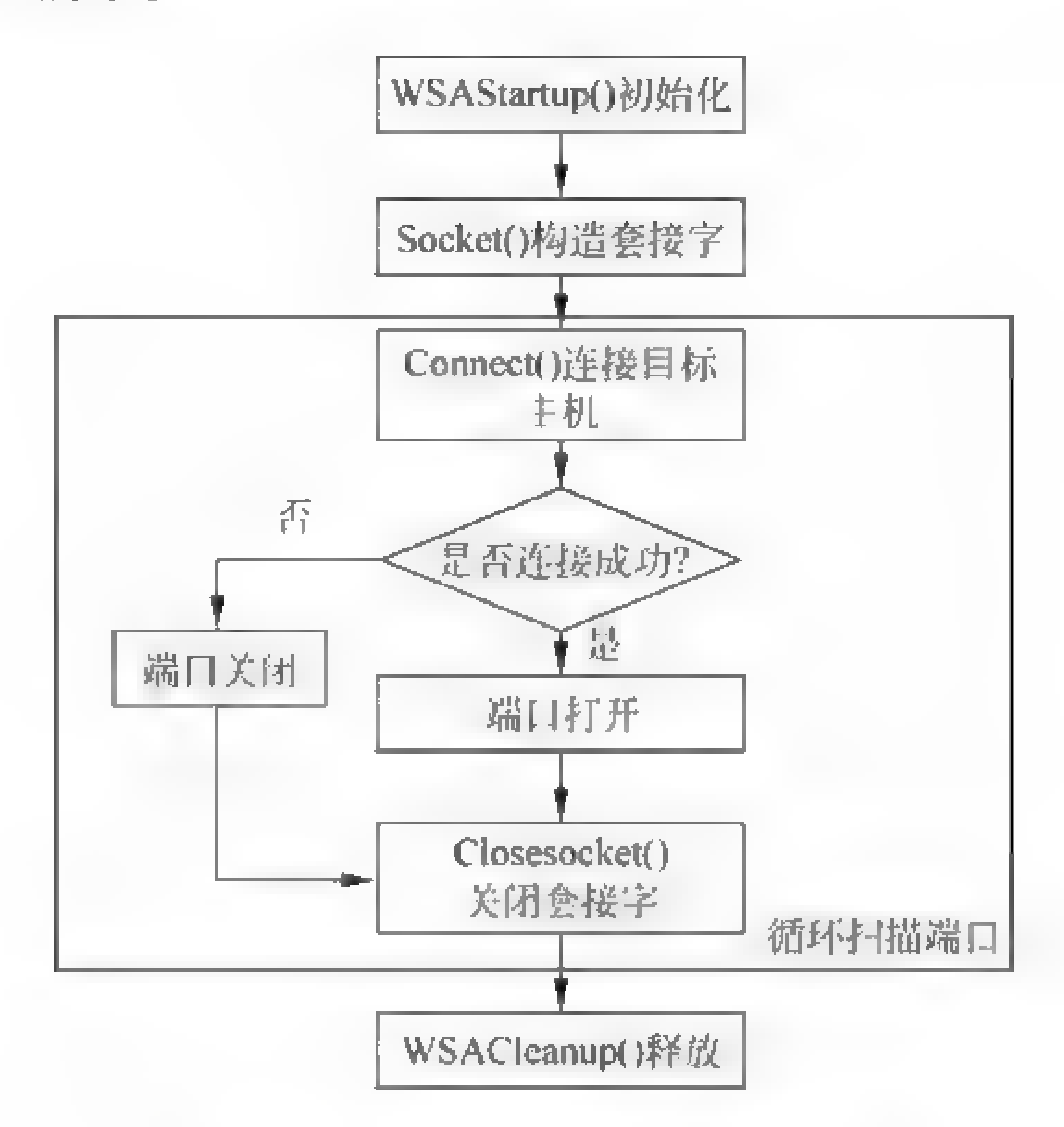


图5-16 TCP连接扫描实现过程

2) TCP连接实现代码

根据上述流程，TCP连接实现代码如下。

```
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include <WinSock2.h>
#pragma comment(lib,"ws2_32.lib")

int main(int argc,char **argv)
{
    char *TargetIPAddr = "119.75.217.109";          //目标IP地址
    unsigned int StartPort = 80;                    //开始端口
    unsigned int EndPort = 90;                      //结束端口
    SOCKET ScanSocket;                              //套接字
    struct sockaddr_in TargetAddr_in;
        int Ret;
        HANDLE hCon = GetStdHandle(STD_OUTPUT_HANDLE);
        //窗口缓冲区信息
        CONSOLE_SCREEN_BUFFER_INFO bInfo;
        //获取窗口缓冲信息
        GetConsoleScreenBufferInfo(hCon, &bInfo);
        WSADATA wsaData;
        //初始化WinSock库
        if ((Ret = WSAStartup(MAKEWORD(2, 1), &wsaData)) != 0)
```

```
        {
            printf("WSAStartup failed with error %d\n", Ret);
            return 0;
        }
        //获取时间
        DWORD dwStart = GetTickCount();
        printf("Target IP: %s\n", TargetIPAddr);
        for (unsigned int i = StartPort; i <= EndPort; i++)
        {
            ScanSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
            if (ScanSocket == INVALID_SOCKET)
            {
                printf("socket failed with error: %d\n", WSAGetLastError());
                return 0;
            }
            //填充地址结构
            TargetAddr_in.sin_family = AF_INET;
            TargetAddr_in.sin_addr.s_addr = inet_addr(TargetIPAddr);
            //设置目标机端口
            TargetAddr_in.sin_port = htons(i);
            if (connect(ScanSocket, (struct sockaddr *)&TargetAddr_in, sizeof(TargetAddr_
in)) == SOCKET_ERROR)
            {
                //连接不成功,端口未开放
                SetConsoleTextAttribute(hCon, 10);
                printf("port %5d close\n", i);
            }
            else
            {
                //连接成功,端口开放
                SetConsoleTextAttribute(hCon, 14);
                printf("port %5d open\n", i);
            }
            //关闭套接字
            if (closesocket(ScanSocket) == SOCKET_ERROR)
            {
                printf("closesocket failed with error %d\n", WSAGetLastError());
                return 0;
            }
        }//endfor
        SetConsoleTextAttribute(hCon, 14);
        printf("\ntime: %dms \n", GetTickCount() - dwStart);
        SetConsoleTextAttribute(hCon, bInfo.wAttributes);
        //释放套接字
        if (WSACleanup() == SOCKET_ERROR)
        {
            printf("WSACleanup failed with error %d\n", WSAGetLastError());
            return 0;
        }
        return 1;
}
```

运行结果如图 5-17 所示。

图 5-17　TCP 扫描运行结果图

在本程序中核心部分是判断是否连接成功，可以比较简单地判断 connect( )函数的返回值状态，如果返回 SOCKET_ERROR 表示执行错误，也就是对方端口是关闭的，不能够正常进行 TCP 的连接；如果返回成功，表示对方端口是开放的。

```
if (connect(ScanSocket,(struct sockaddr * )&TargetAddr_in,
    sizeof(TargetAddr_in)) == SOCEKT_ERROR)
    {
      //连接不成功,端口未开放
        SetConsoleTextAttribute(hCon,10);
        Printf("Port %5d Close\n", i);
    }
    else
    {
     //连接成功,端口开放
      SetConsoleTextAttribute(hCon,14);
      Printf("Port %5d Open\n", i);
      }
```

在判断端口开放之后，不需要进行数据传输，所以马上把端口关闭，然后再进行下一个端口的扫描。

2. TCP SYN 扫描实现

1) TCP SYN 扫描实现流程

TCP SYN 扫描应用非常广泛，是最好的 TCP 扫描形式，其速度很快，效率高，而且也比较安全隐蔽。TCP 扫描是构造含有 SYN 标志的 TCP 数据包，然后判断返回数据包的内容。TCP SYN 扫描流程如图 5-18 所示。

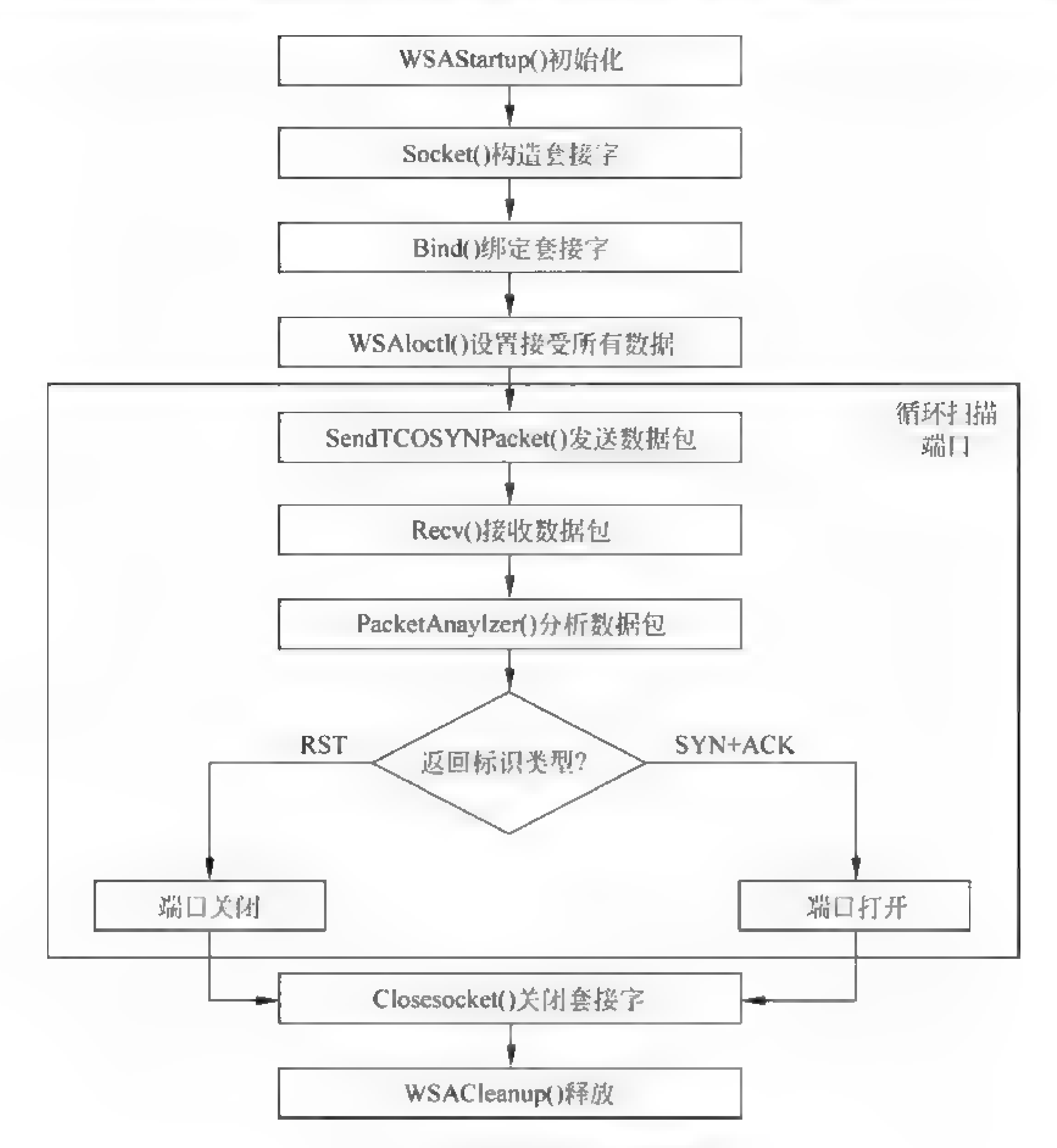


图 5-18　TCP SYN 扫描实现过程

2) TCP SYN 扫描实现代码

基于上述实现流程，TCP SYN 扫描实现代码如下。

```
//定义控制面板应用程序的入口点
//
#include "stdafx.h"
#include "stdio.h"
#include "string.h"
#include "Winsock2.h"
#include <ws2tcpip.h>
#include "mstcpip.h"
#pragma comment(lib,"WS2_32.lib")
char * DestIpAddr = "192.168.1.111";
typedef struct IpHeader
{
    unsigned char Version_HLen;          //头部长度 IP 版本号
    unsigned char TOS;                   //服务类型 TOS
    unsigned short Length;               //总长度
    unsigned short Ident;                //标识
```

```
    unsigned short Flags_Offset;            //标志位
    unsigned char TTL;                      //生存时间 TTL
    unsigned char Protocol;                 //协议
    unsigned short Checksun;                //IP 头部校验和
    unsigned int SourceAddr;                //源 IP 地址
    unsigned int DestinationAddr;           //目标 IP 地址
}Ip_Header;
//TCP 的标志
#define URG 0x20
#define ACK 0x10
#define PSH 0x08
#define RST 0x04
#define SYN 0x02
#define FIN 0x01
#define SIO_RCVALL _WSAIOW(IOC_VENDOR,1)
typedef struct TcpHeader
{
    USHORT SrcPort;                         //16 位源端口
    USHORT DstPort;                         //16 位目标端口
    unsigned int SequenceNum;               //32 位序号
    unsigned int Acknowledgment;            //32 位确认序号
    unsigned char HdrLen;                   //头部长度
    unsigned char Flags;                    //6 位标志位
    USHORT AdvertisedWindow;                //16 位窗口大小
    USHORT Checksum;                        //16 位校验和
    USHORT UrgPtr;                          // 16 位紧急指针
}Tcp_Header;
//函数引用
//分析数据包
int PacketAnalyzer(char * );
//发送数据包
int SendTCPSYNPacket(int);

//主函数
int main()
{
    //开始端口
    int PortStart = 80;
    //结束端口
    int PortEnd = 90;
    //套接字
    SOCKET RecSocket;
    int Result;
    char RecvBuf[65536] = { 0 };
    //定时器的频率
    LARGE_INTEGER nFreq;
    char Name[255];
    //起始获取定时器的值
    LARGE_INTEGER StartTime;
    //终止定时器的值
```

```
    LARGE_INTEGER EndTime;
    HANDLE hCon;
    WSADATA wsaData;
    DWORD dwBufferLen[10];
    DWORD dwBufferInLen = 1;
    DWORD dwBytesReturned = 0;
    struct hostent * pHostent;
    //初始化 SCOKET
    Result = WSAStartup(MAKEWORD(2, 1), &wsaData);
    if (Result == SOCKET_ERROR)
    {
        printf("WSAStartup failed with error % d\n", Result);
        return 0;
    }
    //创建接收数据的套接字
    RecSocket = socket(AF_INET, SOCK_RAW, IPPROTO_IP);
    if (Result == SOCKET_ERROR)
    {
        printf("socket failed with error % d\n", WSAGetLastError());
        closesocket(RecSocket);
        return 0;
    }
    //获取本机 IP 地址
    Result = gethostname(Name, 255);
    if (Result == SOCKET_ERROR)
    {
        printf("gethostname failed with error % d\n", WSAGetLastError());
        closesocket(RecSocket);
        return 0;
    }
    pHostent = (struct hostent * )malloc(sizeof(struct hostent));
    pHostent = gethostbyname(Name);
    SOCKADDR_IN sock;
    sock.sin_family = AF_INET;
    sock.sin_port = htons(5555);
    memcpy(&sock.sin_addr.S_un.S_addr, pHostent->h_addr_list[0], pHostent->h_length);
    //绑定套接字
    Result = bind(RecSocket, (PSOCKADDR)&sock, sizeof(sock));
    if (Result == SOCKET_ERROR)
    {
        printf("bind failed with error % d\n", WSAGetLastError());
        closesocket(RecSocket);
        return 0;
    }
    //设置 SOCK_RAW 位 SIO_RCVALL
     Result = WSAIoctl(RecSocket, SIO_RCVALL, &dwBufferInLen, sizeof(dwBufferInLen),
&dwBufferLen, sizeof(dwBufferLen), &dwBytesReturned, NULL, NULL);
    if (Result == SOCKET_ERROR)
    {
        printf("WSAIoctl failed with error % d\n", WSAGetLastError());
```

```
        closesocket(RecSocket);
        return 0;
    }
    hCon = GetStdHandle(STD_OUTPUT_HANDLE);
    //窗口缓冲区信息
    CONSOLE_SCREEN_BUFFER_INFO bInfo;
    //获取窗口缓冲区信息
    GetConsoleScreenBufferInfo(hCon, &bInfo);
    //获取是否支持精确定时器
    if (QueryPerformanceFrequency(&nFreq))
    {
        //获取定时器的值
        QueryPerformanceCounter(&StartTime);
        //循环扫描每个端口
        for (int p = PortStart; p <= PortEnd; p++)
        {
            //发送构造的 TCPSYN 数据包
            SendTCPSYNPacket(p);
            //循环监听是否有数据包到达
            while (true)
            {
                memset(RecvBuf, 0, sizeof(RecvBuf));
                Result = recv(RecSocket, RecvBuf, sizeof(RecvBuf), 0);
                if (Result == SOCKET_ERROR)
                {
                    printf("recv failed with error %d\n", WSAGetLastError());
                    closesocket(RecSocket);
                    return 0;
                }
                //分析数据包
                Result = PacketAnalyzer(RecvBuf);
                if (Result == 0)
                {
                    continue;
                }
                else
                {
                    break;
                }
            }
            SetConsoleTextAttribute(hCon, 14);
            //获取定时器的值
            QueryPerformanceCounter(&EndTime);
        }
    }
    //计算时间
    LONGLONG fInterval = EndTime.QuadPart - StartTime.QuadPart;
    printf("total time: %fms\n", fInterval * 1000 / (double)nFreq.QuadPart);
    //恢复原来的属性
    SetConsoleTextAttribute(hCon, bInfo.wAttributes);
```

```
    //关闭套接字
    if (closesocket(RecSocket) == SOCKET_ERROR)
    {
        printf("closesocket failed with error %d\n", WSAGetLastError());
        return 0;
    }
    //释放 Winsock
    if (WSACleanup() == SOCKET_ERROR)
    {
        printf("WSACleanup failed with error %d\n", WSAGetLastError());
        return 0;
    }
    return 1;
}
//计算校验和
USHORT checksum(USHORT *buffer, int size)
{
    unsigned long cksum = 0;
    while (size > 1)
    {
        cksum += *buffer++;
        size -= sizeof(USHORT);
    }
    if (size)
    {
        cksum += *(UCHAR*)buffer;
    }
    cksum = (cksum >> 16) + (cksum & 0xffff);
    cksum += (cksum >> 16);
    return(USHORT)(~cksum);
}
//发送 TCP 数据
//为了独立性,协议的数据结构全部定义在内部
int SendTCPSYNPacket(int Port)
{
    typedef struct IpHeader
    {
        u_char Version_HLen;            //头部长度 IP 版本号
        u_char TOS;                     //服务类型 TOS
        short Length;                   //总长度
        short Ident;                    //标识
        short Flags_Offset;             //标志位
        u_char TTL;                     //生存时间 TTL
        u_char Protocol;                //协议
        short Checksum;                 //IP 头部校验和
        unsigned int SourceAddr;        //源 IP 地址
        unsigned int DestinationAddr;   //目标 IP 地址
    }Ip_Header;
    //定义 TCP 伪首部
    typedef struct PsdTcpHeader
```

```
    {
        unsigned long SourceAddr;           //源地址
        unsigned long DestinationAddr;      //目标地址
        char Zero;
        char Protcol;                       //协议类型
        unsigned short TcpLen;              //TCP 长度
    }PSD_Tcp_Header;
    //定义 TCP 头部
    typedef struct tcp_hdr
    {
        USHORT SrcPort;
        USHORT DstPort;
        unsigned int SequenceNum;
        unsigned int Acknowledgment;
        unsigned char HdrLen;
        unsigned char Flags;
        USHORT AdvertisedWindow;
        USHORT Checksum;
        USHORT UrgPtr;
    }Tcp_Header;
    //本机 IP 地址
    struct in_addr localaddr;
    char HostName[255];
    struct hostent * Hostent;
    WSADATA wsaData;
    //发送用的套接字
    SOCKET SendSocket;
    SOCKADDR_IN addr_in;
    //IP 头部
    Ip_Header ipHeader;
    //TCP 头部变量
    Tcp_Header tcpHeader;
    //TCP 伪头部
    PSD_Tcp_Header psdHeader;
    //发送缓冲区
    char szSendBuf[100] = {0};
    BOOL flag;
    int nTimeOver;
    int Result;
    Result = WSAStartup(MAKEWORD(2, 1), &wsaData);
    if (Result == SOCKET_ERROR)
    {
        printf("WSAStartup failed with error %d\n", Result);
        return 0;
    }
    if ((SendSocket = WSASocket(AF_INET, SOCK_RAW, IPPROTO_RAW, NULL, 0, WSA_FLAG_
OVERLAPPED)) == INVALID_SOCKET)
    {
        printf("WSASocket failed with error %d\n\n", WSAGetLastError());
        return false;
```

```
    }
    flag = true;
    if (setsockopt(SendSocket, IPPROTO_IP, IP_HDRINCL, (char * )&flag, sizeof(flag)) == SOCKET_
ERROR)
    {
        printf("setsockopt failed with error % d\n\n", WSAGetLastError());
        return false;
    }
    nTimeOver = 1000 ;
    if (setsockopt(SendSocket, SOL_SOCKET, SO_SNDTIMEO, (char * )&nTimeOver, sizeof(nTimeOver)) ==
SOCKET_ERROR)
    {
        printf("setsockopt failed with error % d\n\n", WSAGetLastError());
        return false;
    }
    addr_in.sin_family = AF_INET;
    //端口号
    addr_in.sin_port = htons(1000);
    //对方 IP
    addr_in.sin_addr.S_un.S_addr = inet_addr(DestIpAddr);
    //获取本地 IP 地址
    Result = gethostname(HostName, 255);
    if (Result == SOCKET_ERROR)
    {
        printf("gethostname failed with error % d\n", WSAGetLastError());
        return 0;
    }
    Hostent = (struct hostent * )malloc(sizeof(struct hostent));
    Hostent = gethostbyname(HostName);
    memcpy(&localaddr, Hostent -> h_addr_list[0], Hostent -> h_length);
    //填充 IP 头部
    //版本和长度
    ipHeader.Version_HLen = (4 << 4 | sizeof(ipHeader) + sizeof(tcpHeader));
    //服务类型
    ipHeader.TOS = 0;
    //总长度
    ipHeader.Length = htons(sizeof(ipHeader) + sizeof(tcpHeader));
    //16 为标识
    ipHeader.Ident = 1;
    //偏移
    ipHeader.Flags_Offset = 0;
    //生存时间
    ipHeader.TTL = 128;
    //协议类型
    ipHeader.Protocol = IPPROTO_TCP;
    //校验和清零
    ipHeader.Checksum = 0;
    //源 IP 地址
    ipHeader.SourceAddr = localaddr.S_un.S_addr;
    //目标 IP 地址
```

```
    ipHeader.DestinationAddr = inet_addr(DestIpAddr);
    //填充 TCP 头部
    //目标端口号
    tcpHeader.DstPort = htons(Port);
    //源端口
    tcpHeader.SrcPort = htons(6666);
    //序列号
    tcpHeader.SequenceNum = htonl(0);
    //确认号
    tcpHeader.Acknowledgment = 0;
    //头部长度
    tcpHeader.HdrLen = (sizeof(tcpHeader) / 4 << 4 | 0);
    //标识 SYN
    tcpHeader.Flags = 2;
    //窗口大小
    tcpHeader.AdvertisedWindow = htons(512);
    //紧急指针
    tcpHeader.UrgPtr = 0;
    //校验和清零
    tcpHeader.Checksum = 0;
    //填充 TCP 伪头部
    //源 IP
    psdHeader.SourceAddr = ipHeader.SourceAddr;
    //目标 IP
    psdHeader.DestinationAddr = ipHeader.DestinationAddr;
    psdHeader.Zero = 0;
    //协议类型
    psdHeader.Protcol = IPPROTO_TCP;
    //TCP 长度
    psdHeader.TcpLen = htons(sizeof(tcpHeader));
    //计算 TCP 校验和
    memcpy(szSendBuf, &psdHeader, sizeof(psdHeader));
    memcpy(szSendBuf + sizeof(psdHeader), &tcpHeader, sizeof(tcpHeader));
    tcpHeader.Checksum = checksum((USHORT *)szSendBuf, sizeof(psdHeader) + sizeof(tcpHeader));
    //计算 IP 校验和
    memcpy(szSendBuf, &ipHeader, sizeof(ipHeader));
    memcpy(szSendBuf + sizeof(ipHeader), &tcpHeader, sizeof(tcpHeader));
    memset(szSendBuf + sizeof(ipHeader) + sizeof(tcpHeader), 0, 4);
    ipHeader.Checksum = checksum((USHORT *)szSendBuf, sizeof(ipHeader) + sizeof(tcpHeader));
    //更新内容
    memcpy(szSendBuf, &ipHeader, sizeof(ipHeader));
    Result = sendto(SendSocket, szSendBuf, sizeof(ipHeader) + sizeof(tcpHeader), 0, (struct
sockaddr *)&addr_in, sizeof(addr_in));
    if (Result == SOCKET_ERROR)
    {
        printf("gethostname failed with error %d\n", WSAGetLastError());
        return 0;
    }
    //关闭套接字
    if (closesocket(SendSocket) == SOCKET_ERROR)
```

```
    {
        printf("closesocket failed with error %d\n", WSAGetLastError());
        return 0;
    }
    //释放 WinSock 库
    if (WSACleanup() == SOCKET_ERROR)
    {
        printf("WSAcleanup failed with error %d\n", WSAGetLastError());
        return 0;
    }
    return 1;
}
//数据包分析
int PacketAnalyzer(char *PacketBuffer)
{
    Ip_Header *pIpheader;
    int iProtocol, iTTL;
    char szSourceIP[16], szDestIP[16];
    SOCKADDR_IN saSource, saDest;
    pIpheader = (Ip_Header *)PacketBuffer;
    HANDLE hCon = GetStdHandle(STD_OUTPUT_HANDLE);
    //窗口缓冲区信息
    CONSOLE_SCREEN_BUFFER_INFO bInfo;
    //获取窗口缓冲区信息
    GetConsoleScreenBufferInfo(hCon, &bInfo);
    iProtocol = pIpheader->Protocol;
    //源 IP 地址
    saSource.sin_addr.s_addr = pIpheader->SourceAddr;
    ::strcpy(szSourceIP, inet_ntoa(saSource.sin_addr));
    //目标 IP 地址
    saDest.sin_addr.s_addr = pIpheader->DestinationAddr;
    ::strcpy(szDestIP, inet_ntoa(saDest.sin_addr));
    iTTL = pIpheader->TTL;
    //计算 IP 长度
    int iIphLen = sizeof(unsigned long) * (pIpheader->Version_HLen & 0x0f);
    //判断是不是 TCP 协议数据
    if (iProtocol == IPPROTO_TCP)
    {
        Tcp_Header *pTcpHeader;
        //读取 TCP 数据内容
        pTcpHeader = (Tcp_Header *)(PacketBuffer + iIphLen);
        if (pIpheader->SourceAddr == inet_addr(DestIpAddr))
        {
            //如果返回值有 RST
            if (pTcpHeader->Flags&RST)
            {
                //端口关闭
                SetConsoleTextAttribute(hCon, 10);
                printf("Port %d Close\n", ntohs(pTcpHeader->SrcPort));
                return 1;
```

```
            }
            //如果返回值含有 SYN + ACK
            else if ((pTcpHeader->Flags&SYN) && (pTcpHeader->Flags&ACK))
            {
                //端口开放
                SetConsoleTextAttribute(hCon, 14);
                printf("Port %d Open\n", ntohs(pTcpHeader->SrcPort));
                return 1;
            }
        }
    }
    //恢复原来的属性
    SetConsoleTextAttribute(hCon, bInfo.wAttributes);
    return 0;
}
```

运行结果如图 5-19 所示。

本程序的核心内容是构造 TCP SYN 数据包，然后计算其校验和，把构造的数据包通过 sedto()发送给远程目标主机，然后通过监听返回的网络包，分析判断数据包的内容。

图 5-19 TCP SYN 扫描运行结果图

```
if (iProtocol == IPPROTO_TCP)
 {
 Tcp_Header   * pTcpHeader;
//读取 TCP 数据内容
 pTcpHeader = (Tcp_Header * )(PacketBuffer + iIphLen);
  if(pIpheader->SourceAddr  == inet_addr(DestIpAddr))
 {
  //如果返回值有 RST
 if(pTcpHeader->Flags&RST)
 {
    //端口关闭
  Set ConsoleTextAttribute(hCon,10);
 printf("Port %d Close\n", ntohs(pTcpHeader->SrcPort));
   return 1;
 }
 //如果返回值含有 SYN_ACK
 else if(pTcpHeader->Flags&SYN)&&(pTcpHeader->Flags * ACK))
{
     //端口开放
     Set ConsoleTextAttribute(hCon,14);
     printf("Port %d Open \n",ntohs(pTcpHeader->SrcPort));
    return 1;
}
}
}
```

首先要捕获的协议数据是 TCP，不是 TCP 数据包就不用分析，然后读取 TCP 数据内容。

```
pTcpHeader = (Tcp_Header * ) (PackerBuffer + ilphLen);
```

读取 TCP 标志位，如果有 RST，表示目标端口是关闭的，如果是 SYN 和 ACK 标志，表示目标端口是开放的。

5.7.3 UDP 扫描实现

1. UDP 扫描实现流程

UDP 扫描比其他的扫描技术简单一些，但扫描过程较难控制，因为打开的端口对扫描探测并不返回一个确认，关闭的端口也并不需要发送回一个错误的数据包。虽然有些主机在打开的 UDP 端口上，会返回一个 UDP 数据包，但这样的机会很少。另外有些主机会在未打开的 UDP 端口上返回一个 ICMP 不可达数据包。由此可以判断哪个 UDP 端口是关闭的。由于 UDP 和 ICMP 数据包都不能保证准确无误地到达目的地，因此这种扫描会有一定的困难。并且由于操作系统对 ICMP 错误消息的产生速率做了限定，所以 UDP 扫描实现起来速度会很慢。UDP 扫描的具体实现过程如图 5-20 所示。

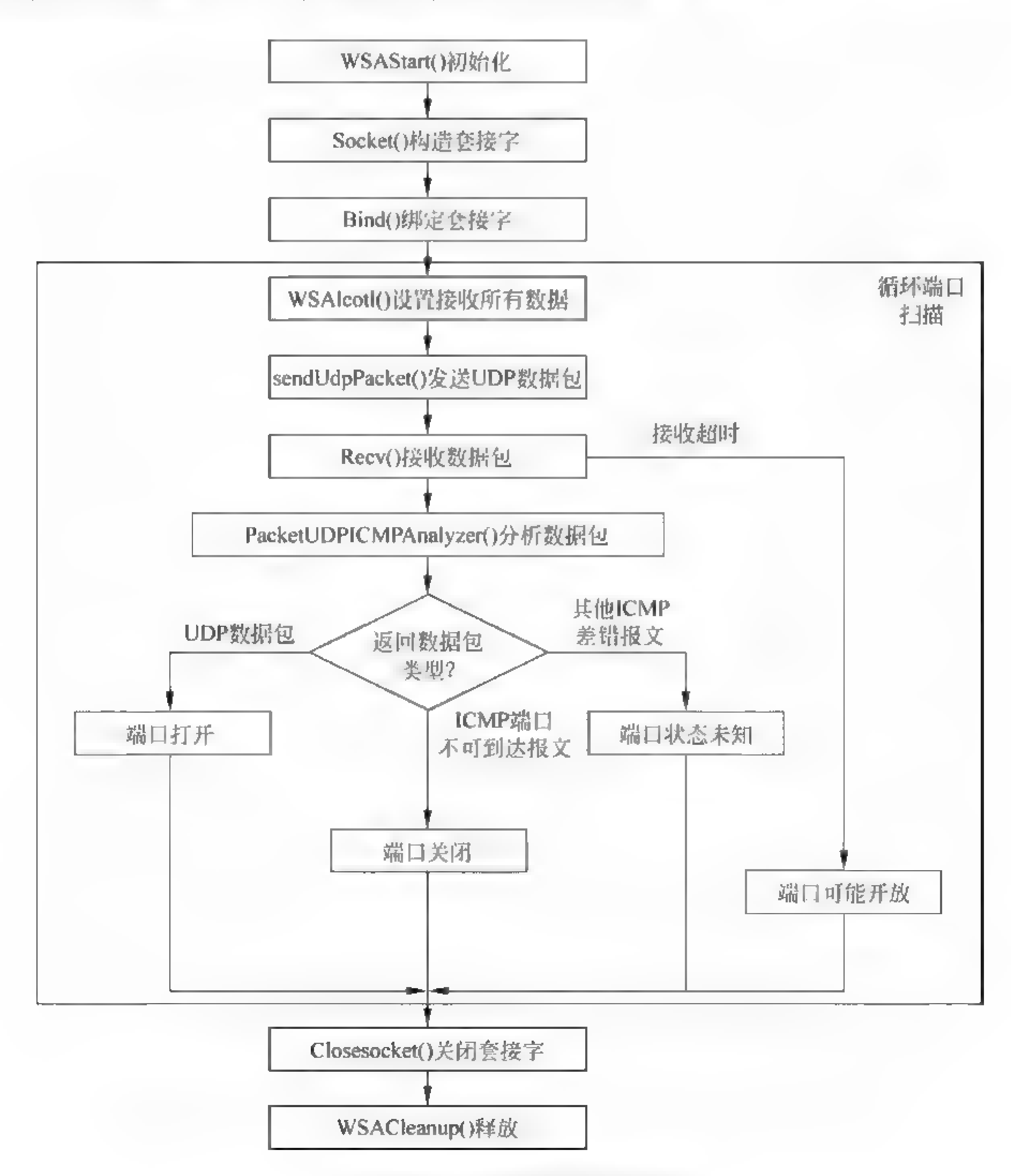


图 5-20 UDP 扫描实现过程

2. UDP 扫描实现代码

```
//定义控制面板应用程序的入口点
//
#include "stdafx.h"
#include "Winsock2.h"
#include <ws2tcpip.h>
#include "mstcpip.h"
#pragma comment(lib,"WS2_32.lib")
int Stop = 0;
//开始端口
int StartPort = 1898;
//结束端口
int EndPort = 1901;
//扫描目标主机的 IP 地址,可以设定需要扫描的机器地址
char *DestIpAddr = "10.171.68.54";
typedef struct IpHeader
{
unsigned char Version_HLen;            //头部长度 IP 版本号
unsigned char TOS;                     //服务类型 TOS
unsigned short Length;                 //总长度
unsigned short Ident;                  //标识
unsigned short Flags_Offset;           //标志位
unsigned char TTL;                     //生存时间
unsigned char Protocol;                //协议
unsigned short Checksum;               //校验和
unsigned int SourceAddr;               //源 IP 地址
unsigned int DestinationAddr;          //目标 IP 地址
}Ip_Header;
//TCP 的标志
#define URG 0x20
#define ACK 0x10
#define PSH 0x08
#define RST 0x04
#define SYN 0x02
#define FIN 0x01
//定义 TCP 头部
typedef struct TcpHeader
{
USHORT SrcPort;                        //16 位源端口
USHORT DstPort;                        //16 位目标端口
unsigned int SequenceNum;              //32 位序号
unsigned int Acknowledgment;           //32 位确认序号
unsigned char HdrLen;                  //头部长度
unsigned char Flags;                   //6 位标志位
USHORT AdvertisedWindow;               //16 位窗口大小
USHORT Checksum;                       //16 位校验和
USHORT UrgPtr;                         //16 位紧急指针
}Tcp_Header;
```

```
typedef struct UdpHeader
{
u_short SrcPort;
u_short DstPort;
u_short Length;
u_short Checksum;
}Udp_Header;
typedef struct IcmpHeader
{
BYTE i_type;                    //8 位类型
BYTE i_code;                    //8 位代码
USHORT i_checksum;              //16 位校验和
USHORT i_id;                    //16 位识别号
USHORT i_sequence;              //16 位报文序列号
}Icmp_Header;
//函数引用
//解析 IP 协议
int PacketUDPICMPAnalyzer(int, char * );
//发送 UDP 数据包
int SendUdpPacket(int, char * );

int main()
{
//套接字
SOCKET RecSocket;
int Result;
char RecBuf[65535] = { 0 };
//定时器的频率
LARGE_INTEGER nFreq;
char Name[255];
//起始获取定时器的值
LARGE_INTEGER StartTime;
//终止定时器的值
LARGE_INTEGER EndTime;
HANDLE hCon;
WSADATA wsaData;
DWORD dwBufferLen[10];
DWORD dwBufferInLen = 1;
DWORD dwBytesReturned = 0;
struct hostent * pHostent;
//初始化 SOCKET
Result = WSAStartup(MAKEWORD(2, 1), &wsaData);
if (Result == SOCKET_ERROR)
{
    printf("WSAStartup failed error %d\n", Result);
    return 0;
}
//创建接收数据的套接字
RecSocket = socket(AF_INET, SOCK_RAW, IPPROTO_IP);
if (Result == SOCKET_ERROR)
```

```
{
    printf("socket failed wih error %d\n", WSAGetLastError());
    closesocket(RecSocket);
    return 0;
}
//获取本机 IP 地址
Result = gethostname(Name, 255);
if (Result == SOCKET_ERROR)
{
    printf("gethostname failed with error %d\n", WSAGetLastError());
    closesocket(RecSocket);
    return 0;
}
pHostent = (struct hostent *)malloc(sizeof(struct hostent));
pHostent = gethostbyname(Name);
SOCKADDR_IN sock;
sock.sin_family = AF_INET;
sock.sin_port = htons(5555);
memcpy(&sock.sin_addr.S_un.S_addr, pHostent->h_addr_list[0], pHostent->h_length);
//绑定套接字
Result = bind(RecSocket, (PSOCKADDR)&sock, sizeof(sock));
if (Result == SOCKET_ERROR)
{
    printf("bind failed with error %d\n", WSAGetLastError());
    closesocket(RecSocket);
    return 0;
}
//设置 SOCK_RAW 为 SIO_RCAVALL,以便接收所有的 IP 包
Result = WSAIoctl(RecSocket, SIO_RCVALL, &dwBufferInLen, sizeof(dwBufferInLen),
&dwBufferLen, sizeof(dwBufferLen), &dwBytesReturned, NULL, NULL);
if (Result == SOCKET_ERROR)
{
    printf("WSAIoctl failed with error %d\n", WSAGetLastError());
    closesocket(RecSocket);
    return 0;
}
//设置超时
int Timeout = 2000;
//设置套接字属性
//设置超时特性
Result = setsockopt(RecSocket, SOL_SOCKET, SO_RCVTIMEO, (char *)&Timeout, sizeof(Timeout));
if (Result == SOCKET_ERROR)
{
    printf("setsockopt failed with error %d\n", WSAGetLastError());
    return 0;
}
hCon = GetStdHandle(STD_OUTPUT_HANDLE);
//获取窗口缓冲区信息
CONSOLE_SCREEN_BUFFER_INFO bInfo;
//获取精确定时器
```

```
GetConsoleScreenBufferInfo(hCon, &bInfo);
if (QueryPerformanceFrequency(&nFreq))
{
    //获取定时器的值
    QueryPerformanceCounter(&StartTime);
    //循环监听是否有数据包达到
    for (int k = StartPort; k <= EndPort; k++)
    {
        SendUdpPacket(k, DestIpAddr);
        memset(RecBuf, 0, sizeof(RecBuf));
        while (1)
        {
            Result = recv(RecSocket, RecBuf, sizeof(RecBuf), 0);
            if (Result == SOCKET_ERROR)
            {
                if (WSAGetLastError() != WSAETIMEDOUT)
                {
                    printf("recvfrom failed error %d\n", WSAGetLastError());
                    return 0;
                }
                else
                {
                    //超时,表示端口可能是开放的
                    printf("Port %d maybe open1.\n", k);
                    break;
                }
            }
            static int number = 0;
            //分析数据包
            Result = PacketUDPICMPAnalyzer(k, RecBuf);
            if (Result == 0)
            {
                //接收数据,但是数据不是想要的数据
                //收到三次以上则跳出本次循环
                number++;
                if (number > 3)
                {
                    //没有确凿证据表明端口是否开放和关闭
                    //端口有可能是开放的
                    printf("Port %d may be open2.\n", k);
                    break;
                }
                else
                {
                    continue;
                }
            }
            else
            {
                //如果返回值是正确的,表明以及确定端口的状态
```

```
                number = 0;
                break;
            }
        }
    }
    SetConsoleTextAttribute(hCon, 14);
    //获取定时器的值
    QueryPerformanceCounter(&EndTime);
}
//计算时间
double flnterval = EndTime.QuadPart - StartTime.QuadPart;
printf("Total Time: %dfms\n", flnterval * 1000 / (double)nFreq.QuadPart);
SetConsoleTextAttribute(hCon, bInfo.wAttributes);          //恢复原来的属性
//关闭套接字
if (closesocket(RecSocket) == SOCKET_ERROR)
{
    printf("closesocket failed with error %d\n", WSAGetLastError());
    return 0;
}
//释放 WinSock 库
if (WSACleanup() == SOCKET_ERROR)
{
    printf("WSACleanup failed with error %d\n", WSAGetLastError());
    return 0;
}
return 1;
}
//计算校验和
USHORT checksum(USHORT * buffer, int size)
{
unsigned long cksum = 0;
while (size > 1)
{
    cksum += *buffer++;
    size -= sizeof(USHORT);
}
if (size)
{
    cksum += *(UCHAR*)buffer;
}
cksum = (cksum >> 16) + (cksum & 0xffff);
cksum += (cksum >> 16);
return(USHORT)(~cksum);
}
//发送 TCP 数据
//为了独立性,协议数据结构全部定义在内
int SendUdpPacket(int Port,char * DstIp)
{
typedef struct IpHeader
{
```

```
    u_char Version_HLen;
    u_char TOS;
    short Length;
    short Ident;
    short Flags_Offset;
    u_char TTL;
    u_char Protocol;
    short Checksum;
    unsigned int SourceAddr;
    unsigned int DestinationAddr;
}Ip_Header;
typedef struct PsdTcpHeader
{
    unsigned int SourceAddr;
    unsigned int DestinationAddr;
    u_char Zero;
    u_char Protcol;
    unsigned short UdpLength;
}PsdTcp_Header;
//定义 UDP 头部
typedef struct UdpHeader
{
    u_short SrcPort;
    u_short DstPort;
    u_short Length;
    u_short Checksum;
}Udp_Header;
//本机 IP 地址
struct in_addr localaddr;
//本机主机名
char HostName[255];
struct hostent * Hostent;
WSADATA wsaData;
//发送用的套接字
SOCKET SendSocket;
SOCKADDR_IN addr_in;
//表示 IP 头部
Ip_Header ipHeader;
//表示 UDP 头部变量
Udp_Header udpHeader;
//表示 UDP 伪头部
PsdTcp_Header psdudpHeader;
//发送缓冲区
char szSendBuf[100] = { 0 };
BOOL flag;
int nTimeOver;
int Result;
Result = WSAStartup(MAKEWORD(2, 1), &wsaData);
if (Result == SOCKET_ERROR)
{
```

```
        printf("WSAStartup failed with error %d\n", Result);
        return 0;
    }
    if ((SendSocket = WSASocket(AF_INET, SOCK_RAW, IPPROTO_UDP, NULL, 0, WSA_FLAG_OVERLAPPED)) ==
    INVALID_SOCKET)
    {
        printf("WSASocket failed with error %d\n\n", WSAGetLastError());
        return false;
    }
    flag = true;
    if (setsockopt(SendSocket, IPPROTO_IP, IP_HDRINCL, (char *)&flag, sizeof(flag)) == SOCKET_ERROR)
    {
        printf("setsockopt failed with error %d\n\n", WSAGetLastError());
        return false;
    }
    nTimeOver = 1000;
    if (setsockopt(SendSocket, SOL_SOCKET, SO_SNDTIMEO, (char * )&nTimeOver, sizeof(nTimeOver)) ==
    SOCKET_ERROR)
    {
        printf("setsockopt failed with error %d\n\n", WSAGetLastError());
        return false;
    }
    addr_in.sin_family = AF_INET;
    //端口号
    addr_in.sin_port = htons(1000);
    //对方的 IP 地址
    addr_in.sin_addr.S_un.S_addr = inet_addr(DestIpAddr);
    //获取本机 IP 地址
    Result = gethostname(HostName, 255);
    if (Result == SOCKET_ERROR)
    {
        printf("gethostname failed with error %d\n", WSAGetLastError());
        return 0;
    }
    Hostent = (struct hostent *)malloc(sizeof(struct hostent));
    Hostent = gethostbyname(HostName);
    memcpy(&localaddr, Hostent->h_addr_list[0], Hostent->h_length);
    //填充 IP 头部
    ipHeader.Version_HLen = (4 << 4 | sizeof(ipHeader) + sizeof(udpHeader));
    ipHeader.TOS = 0;
    ipHeader.Length = htons(sizeof(ipHeader) + sizeof(udpHeader));
    ipHeader.Ident = 1;
    ipHeader.Flags_Offset = 0;
    ipHeader.TTL = 128;
    ipHeader.Protocol = IPPROTO_UDP;
    ipHeader.Checksum = 0;
    ipHeader.SourceAddr = localaddr.S_un.S_addr;
    //inet_addr(myip);                          //设置本地 IP
    ipHeader.DestinationAddr = inet_addr(DestIpAddr);
    //填充 UDP 头部
```

```
udpHeader.DstPort = htons(Port);
udpHeader.SrcPort = htons(6666);                    //源端口号
udpHeader.Length = htons(sizeof(udpHeader));
udpHeader.Checksum = 0;
psdudpHeader.SourceAddr = ipHeader.SourceAddr;
psdudpHeader.DestinationAddr = ipHeader.DestinationAddr;
psdudpHeader.Zero = 0;
psdudpHeader.Protcol = IPPROTO_TCP;
psdudpHeader.UdpLength = htons(sizeof(udpHeader));
memcpy(szSendBuf, &psdudpHeader, sizeof(psdudpHeader));
memcpy(szSendBuf + sizeof(psdudpHeader), &udpHeader, sizeof(udpHeader));
udpHeader.Checksum = checksum((USHORT *)szSendBuf, sizeof(psdudpHeader) + sizeof(udpHeader));
memcpy(szSendBuf, &ipHeader, sizeof(ipHeader));
memcpy(szSendBuf + sizeof(ipHeader), &udpHeader, sizeof(udpHeader));
memcpy(szSendBuf + sizeof(ipHeader) + sizeof(udpHeader), &udpHeader, sizeof(udpHeader));
ipHeader.Checksum = checksum((USHORT *)szSendBuf, sizeof(ipHeader));
memcpy(szSendBuf, &ipHeader, sizeof(ipHeader));
Result = sendto(SendSocket, szSendBuf, sizeof(ipHeader) + sizeof(udpHeader), 0, (struct sockaddr
 *)&addr_in, sizeof(addr_in));
if (Result == SOCKET_ERROR)
{
    printf("gethostname failed with error %d\n", WSAGetLastError());
    return 0;
}
if (closesocket(SendSocket) == SOCKET_ERROR)
{
    printf("closesocket failed with error %d\n", WSAGetLastError());
    return 0;
}
if (WSACleanup() == SOCKET_ERROR)
{
    printf("WSAcleanup failed with error %d\n", WSAGetLastError());
    return 0;
}
return 1;
}
//数据包解析
int PacketUDPICMPAnalyzer(int Port,char *PacketBuffer)
{
Ip_Header *pIpheader;
int iProtocol, iTTL;
char protocolstr[100] = "\0";
char szSourceIP[16], szDestIP[16];
SOCKADDR_IN saSource, saDest;
pIpheader = (Ip_Header *)PacketBuffer;
HANDLE hCon = GetStdHandle(STD_OUTPUT_HANDLE);
HANDLE hCom = GetStdHandle(STD_OUTPUT_HANDLE);
//窗口缓冲区信息
CONSOLE_SCREEN_BUFFER_INFO bInfo;
//获取窗口缓冲区信息
```

```
GetConsoleScreenBufferInfo(hCon, &bInfo);
iProtocol = pIpheader->Protocol;
saSource.sin_addr.s_addr = pIpheader->SourceAddr;
::strcpy(szSourceIP, inet_ntoa(saSource.sin_addr));
saDest.sin_addr.s_addr = pIpheader->DestinationAddr;
::strcpy(szDestIP, inet_ntoa(saDest.sin_addr));
iTTL = pIpheader->TTL;
//计算 IP 长度
int iIphLen = sizeof(unsigned long) * (pIpheader->Version_HLen & 0x0f);
//判断是否是正确的返回数据包
if (pIpheader->SourceAddr == inet_addr(DestIpAddr))
{
    //判断是否是 ICMP 数据
    if (iProtocol == IPPROTO_ICMP)
    {
        //ICMP 头部
        Icmp_Header *icmp;
        //读取 ICMP 数据
        icmp = (Icmp_Header*)(PacketBuffer + sizeof(Ip_Header));
        //判断是否是应答
        if ((icmp->i_type == 3) && (icmp->i_code == 3))
        {
            //确定端口是关闭
            printf("Port %d close\n", Port);
            //返回 1 表示成功
            return 1;
        }
        else
        {
            //端口开放状态未知
            printf("port %d Unknown\n", Port);
            return  1;
        }
    }
    else if (iProtocol == IPPROTO_UDP)
    {
        //如果返回的是 UDP 报文,则端口是开放的
        printf("Port %d Open\n", Port);
        //能够正确判断,成功返回
        return 1;
    }
    else
    {
        //既不是 ICMP 数据又不是 UDP 数据,则表示不是想要的数据
        //返回错误
        return 0;
    }
}
else
{
```

```
        //不是正确的返回数据包
        return 0;
    }
    //恢复原来的属性
    SetConsoleTextAttribute(hCon, bInfo.wAttributes);
    return 1;
    }
```

调试方法：首先了解自己计算机的 TCP、UDP 端口连接情况，在命令窗口中输入 netstat -an 可以看到如图 5-21 所示信息，这是部分信息，可以知道自己的 IP 地址，选取 1900 端口，设置 ip＝10.171.68.54，起始端口为 1898，结束端口 1901。

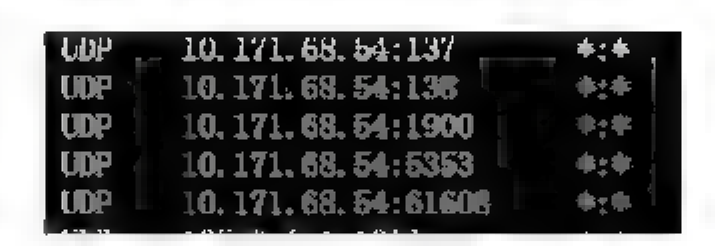


图 5-21　端口信息

UDP 扫描运行结果如图 5-22 所示。



图 5-22　UDP 扫描运行结果图

5.7.4　木马扫描实现

典型的木马是基于客户端和服务器端的模式，在服务器端必须有一个监听端口，扫描此端口的开放状况，可以在某种程度上检测主机是否被植入某种木马。然而现今的木马形式很多，有些木马并不需要特别的监听端口。它们可以通过各种途径与客户端通信，例如可以借助已知的进程端口，这样就不会被轻易发现。

以下以冰河作为例子。冰河其实是一个远程控制程序，但如果把它用来进行入侵破坏活动，就变成了木马。冰河的默认监听端口为 7626，可以使用 netstat 命令查看。其实现代码如下。

```
//定义控制面板应用程序的入口点
//
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include <winsock2.h>
#pragma comment(lib,"ws2_32.lib")
//把 IP 字符串形式转化为 4 个十进制数字
void  IpToNum(char *straddr, unsigned char byteaddr[4])
{
    char temp[16];
```

```
        char * k;
        memcpy(temp,straddr,strlen(straddr));
        temp[strlen(straddr)] = '\0';
        char * m = temp;
        for (int i = 0; i < 3; i++)
        {
            k = strchr(m, '.');
            char addr[4];
            for (int j = 0; j < (strlen(m) - strlen(k)); j++)
            {
                addr[j] = m[j];
                addr[j] = '\0';
            }
            byteaddr[i] = atoi(addr);
            k = k + 1;
            m = k;
        }
        byteaddr[3] = atoi(m);
    }
    int main(int argc, char ** argv)
    {
        //起始目标 IP 地址
        char * TargetIpAddrStart = "192.168.1.2";
        //终止目标 IP 地址
        char * TargetIpAddrEnd = "192.168.1.4";
        //开始端口
        unsigned int StartPort = 7626;
        //结束端口
        unsigned int EndPort = 7626;
        //套接字
        SOCKET ScanSocket;
        //地址结构
        struct sockaddr_in TargetAddr_in;
        int Ret;
        HANDLE hCon = GetStdHandle(STD_OUTPUT_HANDLE);
        //窗口缓冲区信息
        CONSOLE_SCREEN_BUFFER_INFO bInfo;
        //获取窗口缓冲区信息
        GetConsoleScreenBufferInfo(hCon,&bInfo);
        WSADATA wsaData;
        //初始化 WinSock 库
        if ((Ret = WSAStartup(MAKEWORD(2,1),&wsaData))!= 0)
        {
            printf("WSAStrartup failed with error %d\n",Ret);
            return 0;
        }
        //获取时间
        DWORD dwStart = GetTickCount();
        BYTE ip[4];
        IpToNum(TargetIpAddrStart,ip);
```

```
int IpAddrStart = ip[3];
IpToNum(TargetIpAddrEnd,ip);
int IpAddrEnd = ip[3];
for (int addr = IpAddrStart; addr <= IpAddrEnd; addr++)
{
    for (unsigned int i = StartPort; i <= EndPort; i++)
    {
        //创建连接端口的套接字
        ScanSocket = socket(AF_INET,
            SOCK_STREAM, IPPROTO_TCP);
        if (ScanSocket == INVALID_SOCKET)
        {
            printf("socket failed with error : %d\n", WSAGetLastError());
            return 0;
        }
        //填充地址结构
        TargetAddr_in.sin_family = AF_INET;
        ip[3] = addr;
        char CurrentIp[100] = "\0";
        sprintf(CurrentIp," %d. %d. %d. %d",ip[0],ip[1],ip[2],ip[3]);
        TargetAddr_in.sin_port = htons(i);
        if (connect(ScanSocket,(struct sockaddr * )&TargetAddr_in,
            sizeof(TargetAddr_in)) == SOCKET_ERROR)
        {
            //连接不成功,端口未开放
            SetConsoleTextAttribute(hCon,10);
            printf(" %s Binghe no found.\n",CurrentIp);
        }
        else
        {
            //链接成功,端口开放
            SetConsoleTextAttribute(hCon,14);
            printf(" %s Binghe may be up.\n",CurrentIp);
        }
        //关闭套接字
        if (closesocket(ScanSocket) == SOCKET_ERROR)
        {
            printf("closesocket failed with error %d\n",
                WSAGetLastError());
            return 0;
        }
    }//end for
}//end for
SetConsoleTextAttribute(hCon,14);
printf("\ntime: %dms\n",GetTickCount() - dwStart);
SetConsoleTextAttribute(hCon, bInfo.wAttributes);          //恢复原来的属性
//释放 WinSock 库
if (WSACleanup() == SOCKET_ERROR)
{
    printf("WSACleanup failed with erroe %d\n",WSAGetLastError());
```

```
        return 0;
    }
    return 1;
}
```

运行结果如图 5-23 所示。



图 5-23 木马扫描运行结果图

5.7.5 隐秘扫描实现

TCP connect( )扫描及 TCP SYN 扫描都容易被防火墙等网络安全防护软件察觉，所以属于非隐秘扫描。相反，如果使用 FIN 标志，则不会被轻易发现。这些能够隐藏扫描行为的方法被称为隐秘扫描方法，而 TCP FIN 扫描是其中的一种。TCP FIN 扫描会发送一个 FIN 数据包给目标主机端口，如果端口是关闭的状态，目标主机会采用适当的 RST 数据包返回；如果端口是打开的，目标主机则会忽略 FIN 数据包的回复。当然，在某些特殊情况下，某些主机系统对于 FIN 数据包，无论端口打开还是关闭都返回 RST 数据包，这样 FIN 扫描就会失去作用。

隐秘扫描代码实现如下。

```
//定义控制面板应用程序的入口点
//
#include "stdafx.h"
#include "stdio.h"
#include "Winsock2.h"
#include <ws2tcpip.h>
#include "mstcpip.h"
#pragma comment(lib,"WS2_32.lib")
//判断是否结束循环
int Stop = 0;
//开始端口
int PortStart = 80;
//结束端口
int PortEnd = 81;
//扫描目标主机的 IP 地址，可以设定需要扫描的机器地址
char * DestIpAddr = "10.171.68.219";
//定义 IP 头部
typedef struct IpHeader
{
//头部长度 IP 版本号
unsigned char Version_HLen;
//服务类型 TOS
unsigned char TOS;
```

```
//总长度
unsigned short Length;
//标识
unsigned short Ident;
//标志位
unsigned short Flags_Offset;
//生存时间 TTL
unsigned char TTL;
//协议
unsigned char Protocol;
//IP 头部校验和
unsigned short Checksum;
//源 IP 地址
unsigned int SourceAddr;
//目标 IP 地址
unsigned int DestinationAddr;
}Ip_Header;
//TCP 的标志
#define URG 0x20
#define ACK 0x10
#define PSH 0x08
#define RST 0x04
#define SYN 0x02
#define FIN 0x01
//定义 TCP 头部
typedef struct TcpHeader
{
//16 位源端口
USHORT SrcPort;
//16 位目标端口
USHORT DstPort;
//32 位序号
unsigned int SequenceNum;
//32 位确认序号
unsigned int Acknowledgment;
//头部长度
unsigned char HdrLen;
//6 位标志位
unsigned char Flags;
//16 位窗口大小
USHORT AdvertisedWindow;
//16 校验和
USHORT Checksum;
//16 位紧急指针
USHORT UrgPtr;
}Tcp_Header;
//函数引用
//解析 IP 协议
int PacketAnalyzer(char * );
//发送数据包
```

```
DWORD WINAPI Send_Net_Packet(LPVOID no);
//主函数
int main(int argc, char * * argv)
{
//建立一个线程句柄
HANDLE Thread;
//线程 ID
DWORD ThreadId;
//套接字
SOCKET RecSocket;
int Result;
char RecvBuf[65535] = { 0 };
//定时器的频率
LARGE_INTEGER nFreq;
char Name[255];
//起始获取定时器的值
LARGE_INTEGER StartTime;
//终止定时器的值
LARGE_INTEGER EndTime;
HANDLE hCon;
WSADATA wsaData;
DWORD dwBufferLen[10];
DWORD dwBufferInLen = 1;
DWORD dwBytesReturned = 0;
struct hostent * pHostent;
//初始化 SOCKET
Result = WSAStartup(MAKEWORD(2, 1), &wsaData);
if ((Result == SOCKET_ERROR))
{
    printf("WSAStartup failed with error % d\n",Result);
    return 0;
}
//创建接收数据的套接字
RecSocket = socket(AF_INET,SOCK_RAW,IPPROTO_IP);
if (Result == SOCKET_ERROR)
{
    printf("WSASTARTup failed with error % d\n", WSAGetLastError());
    closesocket(RecSocket);
    return 0;
}
//获取本机 IP 地址
Result = gethostname(Name,255 );
if (Result == SOCKET_ERROR)
{
    printf("gethostname failed with error % d\n",WSAGetLastError());
    closesocket(RecSocket);
    return 0;
}
pHostent = (struct hostent * )malloc(sizeof(struct hostent));
pHostent = gethostbyname(Name);
```

```
SOCKADDR_IN sock;
sock.sin_family = AF_INET;
sock.sin_port = htons(5555);
memcpy(&sock.sin_addr.S_un.S_addr,
    pHostent->h_addr_list[0],pHostent->h_length);
//绑定套接字
Result = bind(RecSocket,(PSOCKADDR)&sock,sizeof(sock));
if (Result == SOCKET_ERROR)
{
    printf("bind failed with error %d\n", WSAGetLastError());
    closesocket(RecSocket);
    return 0;
}
//设置 SOCK_RAW 为 SIO_RCALL,以便接收所有的 IP 包
Result = WSAIoctl(
    RecSocket,
    SIO_RCVALL,                          //接收所有的 IP 数据包
    &dwBufferInLen,
    sizeof(dwBufferInLen),
    &dwBufferLen,
    sizeof(dwBufferLen),
    &dwBytesReturned, NULL, NULL);
if (Result == SOCKET_ERROR)
{
    printf("WSAIoctl failed with error %d\n", WSAGetLastError());
    closesocket(RecSocket);
    return 0;
}
//创建一个线程用来发送数据包
Thread = CreateThread(NULL,0,Send_Net_Packet, //回调函数
    NULL,0,&ThreadId);
if (Thread==NULL)
{
    printf("CreateThread for Send_Net_Packet Error. %d",GetLastError());
    return 0;
}
hCon = GetStdHandle(STD_OUTPUT_HANDLE);
//窗体缓冲区信息
CONSOLE_SCREEN_BUFFER_INFO bInfo;
//获取窗口缓冲区信息
GetConsoleScreenBufferInfo(hCon,&bInfo);
//获取是否支持精确定时器
if (QueryPerformanceFrequency(&nFreq))
{
    //获取定时器的值
    QueryPerformanceCounter(&StartTime);
    //循环监听是否有数据包到达
    while (true)
    {
        memset(RecvBuf,0,sizeof(RecvBuf));
```

```
        Result = recv(RecSocket, RecvBuf, sizeof(RecvBuf), 0);
        if (Result == SOCKET_ERROR)
        {
            printf("recv failed with error %d\n",WSAGetLastError());
            closesocket(RecSocket);
            return 0;
        }
        //分析数据包
        Result = PacketAnalyzer(RecvBuf);
        if (Result == 0)
        {
            printf("PacketAnalyzer failed with error %d\n",Result);
            closesocket(RecSocket);
            return 0;
        }
        //是否停止
        if (Stop == 1)
        {
            break;
        }
    }
    SetConsoleTextAttribute(hCon,14);
    //获取定时器的值
    QueryPerformanceCounter(&EndTime);
}
//计算时间花费多少秒
double fInterval = EndTime.QuadPart - StartTime.QuadPart;
printf("Total Time:%fms\n",fInterval * 1000/(double)nFreq.QuadPart );
SetConsoleTextAttribute(hCon,bInfo.wAttributes);            //恢复原来的属性
//关闭套接字
if (closesocket(RecSocket) == SOCKET_ERROR)
{
    printf("closesocket failed with error %d\n",WSAGetLastError());
    return 0;
}
//释放 WinSock 库
if (WSACleanup() == SOCKET_ERROR)
{
    printf("WSACleanup failed with error %d\n",WSAGetLastError());
    return 0;
}
return 1;
}
//计算校验和
USHORT checksum(USHORT * buffer, int size)
{
unsigned long cksum = 0;
while (size>1)
{
    cksum +=  *buffer++;
```

```
    size = sizeof(USHORT);
}
if (size)
{
    cksum +=  *(UCHAR*)buffer;
}
cksum = (cksum>>16) + (cksum &0xffff);
cksum += (cksum>>16);
return(USHORT)(~cksum);
}
//发送 TCP 数据
//为了独立性,协议的数据结构全部定义在内部
DWORD WINAPI Send_Net_Packet(LPVOID no)
{
//定义 IP 头部
typedef struct IpHeader
{
    u_char Version_HLen;                //头部长度 IP 版本号
    u_char TOS;                         //服务类型 TOS
    short Length;                       //总长度
    short Ident;                        //标识
    short Flags_Offset;                 //标志位
    u_char TTL;                         //生存时间 TTL
    u_char Protocol;                    //协议
    short Checksum;                     //IP 头部校验和
    unsigned int SourceAddr;            //源 IP 地址
    unsigned int DestinationAddr;       //目标 IP 地址
}; Ip_Header;
//定义 TCP 伪头部
typedef struct tsd_hdr
{
    unsigned long saddr;                //源地址
    unsigned long daddr;                //目标地址
    char mbz;
    char ptcl;                          //协议类型
    unsigned short tcpl;                //TCP 长度
}PSD_Tcp_Header;
//定义 TCP 首部
typedef struct tcp_hdr
{
    USHORT SrcPort;                     //16 位源端口
    USHORT DstPort;                     //16 位目标端口
    unsigned int SequenceNum;           //32 位序列号
    unsigned int Acknowledgment;        //32 位确认号
    unsigned char HdrLen;               //头部长度
    unsigned char Flags;                //6 位标志位
    USHORT AdvertisedWindow;            //16 位窗口大小
    USHORT Checksum;                    //16 位校验和
    USHORT UrgPtr;                      //16 位紧急数据偏移量
}Tcp_Header;
```

```
//本机 IP 地址
struct in_addr localaddr;
//本地主机名
char HostName[255];
struct hostent * Hostent;
WSADATA wsaData;
//发送用的套接字
SOCKET SendSocket;
SOCKADDR_IN addr_in;
//IP 头部
Ip_Header ipHeader;
//TCP 头部变量
Tcp_Header tcpHeader;
//TCP 伪首部
PSD_Tcp_Header psdHeader;
//发送缓冲区
char szSendBuf[100] = { 0 };
BOOL flag;
int nTimeOver;
int Result;
//初始化 SOCKET
Result = WSAStartup(MAKEWORD(2, 1), &wsaData);
if (Result == SOCKET_ERROR)
{
    printf("WSAStartup failed with error %d\n", Result);
    return 0;
}
if ((SendSocket = WSASocket(AF_INET, SOCK_RAW,
    IPPROTO_RAW, NULL, 0, WSA_FLAG_OVERLAPPED)) == INVALID_SOCKET)
{
    printf("WSASocket failed with error %d\n\n", WSAGetLastError());
    return 0;
}
flag = true;
if (setsockopt(SendSocket, IPPROTO_IP, IP_HDRINCL,
    (char * )&flag, sizeof(flag)) == SOCKET_ERROR)
{
    printf("setsockopt failed with error %d\n\n", WSAGetLastError());
    return false;
}
nTimeOver = 1000;
if (setsockopt(SendSocket, SOL_SOCKET, SO_SNDTIMEO,
    (CHAR * )&nTimeOver, sizeof(nTimeOver)) == SOCKET_ERROR)
{
    printf("setsockopt failed with error %d\n\n", WSAGetLastError());
    return false;
}
addr_in.sin_family = AF_INET;
//端口号
//addr_in.sin_family = AF_INET;
```

```
//端口号
addr_in.sin_port = htons(1000);
//对方 IP
addr_in.sin_addr.S_un.S_addr = inet_addr(DestIpAddr);
//获取本机的 IP 地址
Result = gethostname(HostName, 255);
if (Result == SOCKET_ERROR)
{
    printf("gethostname failed with error %d\n", WSAGetLastError());
    return 0;
}
Hostent = (struct  hostent *)malloc(sizeof(struct hostent));
Hostent = gethostbyname(HostName);
memcpy(&localaddr, Hostent->h_addr_list[0], Hostent->h_length);
//填充 IP 头部
ipHeader.Version_HLen = (4 << 4 | sizeof(ipHeader) / sizeof(unsigned long));
ipHeader.TOS = 0;
ipHeader.Length = htons(sizeof(ipHeader) + sizeof(tcpHeader));
ipHeader.Ident = 1;
ipHeader.Flags_Offset = 0;
ipHeader.TTL = 128;
ipHeader.Protocol = IPPROTO_TCP;
ipHeader.Checksum = 0;
ipHeader.SourceAddr = localaddr.S_un.S_addr;
//inet_addr(myip);                        //设置本地 IP 地址
ipHeader.DestinationAddr = inet_addr(DestIpAddr);
//设定要访问的 IP 地址,对方的 IP 地址
for (int p = PortStart; p <= PortEnd; p++)
{
    //填充 TCP 头部
    tcpHeader.DstPort = htons(p);
    tcpHeader.SrcPort = htons(6666);        //源端口号
    tcpHeader.SequenceNum = htonl(0x12345678);
    tcpHeader.Acknowledgment = 0;
    tcpHeader.HdrLen = (sizeof(tcpHeader) / 4 << 4 | 0);
    tcpHeader.Flags = 1;                    //SYN
    tcpHeader.AdvertisedWindow = htons(512);
    tcpHeader.UrgPtr = 0;
    tcpHeader.Checksum = 0;
    psdHeader.saddr = ipHeader.SourceAddr;
    psdHeader.daddr = ipHeader.DestinationAddr;
    psdHeader.mbz = 0;
    psdHeader.ptcl = IPPROTO_TCP;
    psdHeader.tcpl = htons(sizeof(tcpHeader));
    //计算校验和
    memcpy(szSendBuf, &psdHeader, sizeof(psdHeader));
    memcpy(szSendBuf + sizeof(psdHeader), &tcpHeader, sizeof(tcpHeader));
    tcpHeader.Checksum = checksum((USHORT *)szSendBuf,
        sizeof(psdHeader) + sizeof(tcpHeader));
    memcpy(szSendBuf, &ipHeader, sizeof(ipHeader));
```

```
    memcpy(szSendBuf + sizeof(ipHeader), &tcpHeader, sizeof(tcpHeader));
    memset(szSendBuf + sizeof(ipHeader) + sizeof(ipHeader), 0, 4);
    ipHeader.Checksum = checksum((USHORT *)szSendBuf,
        sizeof(ipHeader) + sizeof(tcpHeader));
    memcpy(szSendBuf, &ipHeader, sizeof(ipHeader));
    Result = sendto(SendSocket, szSendBuf,
        sizeof(ipHeader) + sizeof(tcpHeader), 0,
        (struct sockaddr *)&addr_in, sizeof(addr_in));
    if (Result == SOCKET_ERROR)
    {
        printf("gethostname failed with error %d\n", WSAGetLastError());
        return 0;
    }
}//end for
//关闭套接字
if (closesocket(SendSocket) == SOCKET_ERROR)
{
    printf("closesocket failed with error %d\n", WSAGetLastError());
    return 0;
}
//释放 WinSock 库
if (WSACleanup() == SOCKET_ERROR)
{
    printf("closesocket failed with error %d\n", WSAGetLastError());
    return 0;
}//
return 1;
}
//解析数据包
int PacketAnalyzer(char *PacketBuffer)
{
Ip_Header *pIpheader;
int iProtocol, iTTL;
char protocolstr[100] = "\0";
char szSourceIP[16], szDestIP[16];
SOCKADDR_IN saSource, saDest;
pIpheader = (Ip_Header *)PacketBuffer;
HANDLE hCon = GetStdHandle(STD_OUTPUT_HANDLE);
//窗口缓冲信息
CONSOLE_SCREEN_BUFFER_INFO bInfo;
//获取窗口缓冲区信息
GetConsoleScreenBufferInfo(hCon, &bInfo);
iProtocol = pIpheader->Protocol;
//Check Soure IP
saSource.sin_addr.s_addr = pIpheader->SourceAddr;
::strcpy(szSourceIP, inet_ntoa(saSource.sin_addr));
//Check Dest IP
saDest.sin_addr.s_addr = pIpheader->DestinationAddr;
::strcpy(szDestIP, inet_ntoa(saDest.sin_addr));
//TTL
```

```
iTTL = pIpheader->TTL;
//计算 IP 长度
int iIphLen = sizeof(unsigned long) * (pIpheader->Version_HLen & 0x0f);
//判断是否是 TCP 数据
if (iProtocol == IPPROTO_TCP)
{
    Tcp_Header * pTcpHeader;
    pTcpHeader = (Tcp_Header * )(PacketBuffer + iIphLen);
    if (pIpheader->SourceAddr == inet_addr(DestIpAddr))
    {
        if (pTcpHeader->Flags &RST)
        {
            SetConsoleTextAttribute(hCon, 10);
            printf("Port %d Close\n", ntohs(pTcpHeader->SrcPort));
        }
        else
        {
            SetConsoleTextAttribute(hCon, 14);
            printf("Port %d Open\n", ntohs(pTcpHeader->SrcPort));
        }
        if (ntohs(pTcpHeader->SrcPort) == PortEnd)
        {
            Stop = 1;
        }
    }
}
//恢复原来的属性
SetConsoleTextAttribute(hCon, bInfo.wAttributes);
return 1;
}
```

运行结果如图 5-24 所示。



图 5-24　隐秘扫描运行结果图

小　　结

网络扫描器是一种重要的网络安全监测设备，也是黑客进行网络攻击的重要工具之一。掌握端口扫描基本工作原理与软件设计方法是信息安全专业人员必备的技能，因此本章针对网络端口扫描器的实现方法展开了研究，介绍了网络端口扫描器的基本结构、工作原理与设计方法，重点介绍了 ICMP 扫描、TCP connect()扫描、TCP SYN 扫描、UDP 扫描等的基本工作原理及实现方法，并给出了具体代码，在此基础上，使学习者能掌握编写端口扫描程

序的方法，并达到自行设计实现端口扫描程序的目的。

思　考　题

1. 什么是端口及端口扫描？

2. TCP扫描包括哪几类？不同类之间如何区别？

3. 编写端口扫描程序，实现TCP connect()扫描、TCP SYN扫描、TCP FIN扫描及UDP扫描等4种基本扫描方法。

4. 设计并实现ping程序，探测主机是否可达。

第6章　防火墙设计与实现

网络上充斥着各种病毒、木马以及针对主机漏洞的攻击，如何使网络主机能有效抵御各种非法入侵，保证重要数据的机密性和安全性已成为当今网络上一个亟待解决的问题。防火墙就是保护网络资源的重要手段之一。本章将介绍防火墙的基本工作原理，研究防火墙系统的设计与软件编程的方法。

6.1　防火墙技术

6.1.1　防火墙概念

网络防火墙技术是一种用来加强网络之间访问控制的特殊网络互连设备，它可以防止外部网络用户以非法手段进入内部网络并访问资源，保护内部网络操作环境。防火墙按照一定的安全策略对两个或多个网络之间传输的数据包如链接方式来实施检查，以决定网络之间的通信是否被允许，并监视网络运行状态。它是计算机网络中的边境检查站，如图6-1所示，受防火墙保护的是内部网络，而防火墙是部署在两个网络之间的一个或一组部件，要求所有进出内部网络的数据流都通过它，并根据安全策略进行检查，只有符合安全策略、被授权的数据流才可以通过，由此保护内部网络安全。它主要是用来阻止外部网络对内部网络的侵扰，是一种逻辑隔离部件，而不是物理隔离部件。

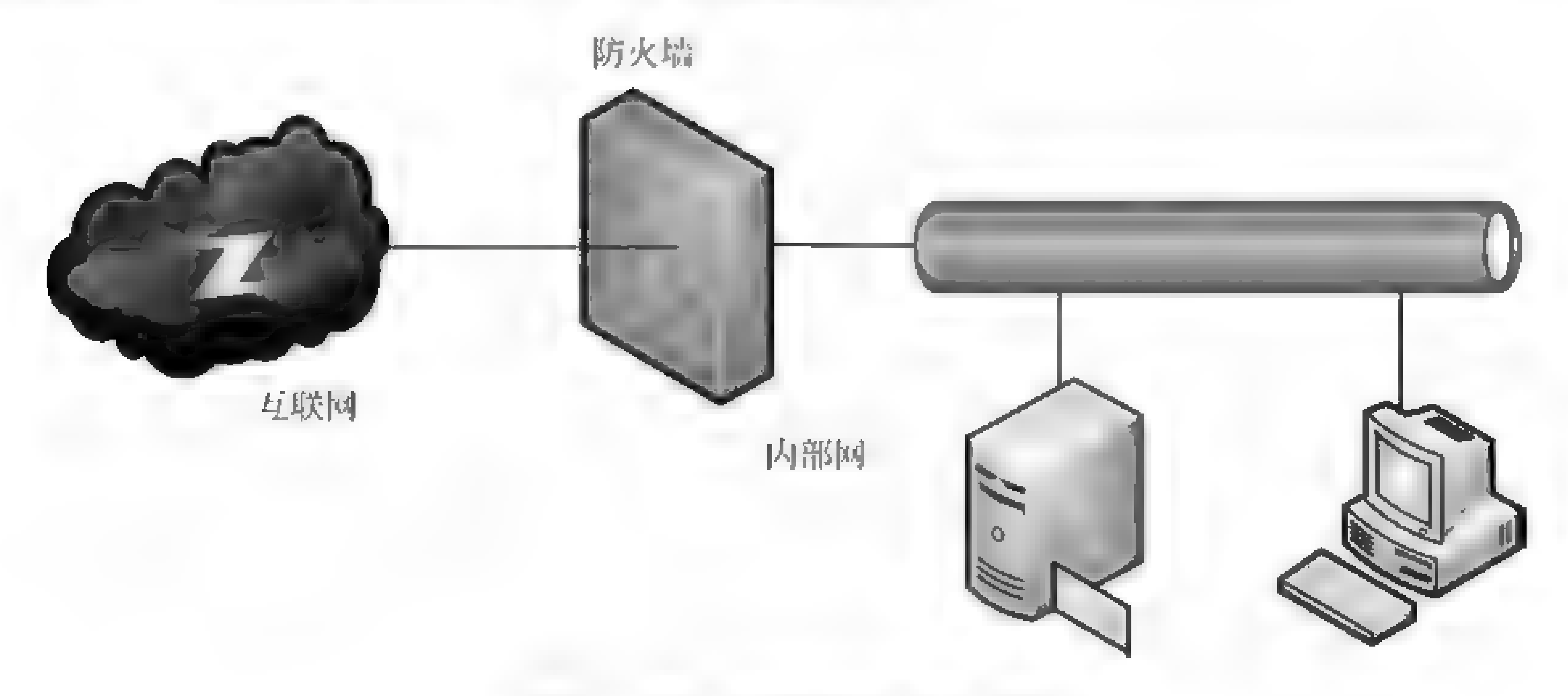


图6-1　防火墙在网络中的位置

自从1986年美国Digital公司在Internet上安装了全球第一个商用防火墙系统，在提出了防火墙的概念后，防火墙技术得到了飞速的发展，国内外已有数十家公司推出了功能各不相同的防火墙产品系列。

1. 防火墙的防护机制

防火墙作为计算机网络中的边境检查站，被部署在网络的边界，在内部网络与外部网络

之间形成隔离，防范外部网络对内部网络的威胁，起到一种边界保护的作用。但由于内部网络的相互访问没有穿越防火墙，所以防火墙无法对此进行控制。防火墙要起到边界保护的作用，要求做到如下几点。

(1) 所有进出内部网络的通信，都必须经过防火墙。

防火墙作为网络边界的安全防护设备，其发挥作用的前提是能够对进出内部网络的所有通信进行检查、控制，如果在受保护的网络内通过拨号上网，该通信绕过了防火墙的检查，将使防火墙失去防护作用。

(2) 所有通过防火墙的通信，都必须经过安全策略的过滤。

即使所有进出内部网络的通信都经过了防火墙，但如果对这些通信不按照安全策略进行检查，或者安全策略的配置漏洞百出、自相矛盾，则防火墙将形同虚设，无法起到应有的防护作用。

(3) 防火墙本身是安全可靠的。

虽然防火墙对所有进出内部网络的通信，按照安全策略都进行了严格的检查，但如果防火墙自身存在安全漏洞，那么黑客就可以通过防火墙的安全漏洞，控制甚至摧毁防火墙。

2. 防火墙的形态

防火墙的访问控制通过一组特别的安全部件实现，其形态有以下几种。

(1) 纯软件。防火墙是运行在通用计算机上的纯软件，简单易用，配置灵活，但因底层操作系统是一个通用型的系统，其数据处理能力、安全性能水平都比较低。

(2) 纯硬件。为解决纯软件防火墙的不足，设计人员将防火墙软件固化在专门设计的硬件上，数据处理能力与安全性能水平都得到很大的提高。但因来自网络的威胁不断变化，防火墙的安全策略、配置等也需要经常进行调整，而纯硬件防火墙的调整非常困难。

(3) 软硬件结合。结合上述两种防火墙的优点，针对防火墙的特殊要求，对硬件、操作系统进行裁减，设计、开发出防火墙专用的硬件、安全操作系统平台，然后在此平台上运行防火墙软件。

在实际应用中，上述三种形态的防火墙可以根据各自的特点灵活应用于不同安全要求的场合，如纯软件防火墙可以应用于个人主机上，纯硬件防火墙可以应用于数据处理性能要求高、安全策略比较稳定的场合等。

3. 防火墙的功能

防火墙是一种网络边界保护型的安全设备，为了达到安全保护内部网络的目的，一般具有如下功能。

(1) 访问控制。这是防火墙最基本、最重要的功能。防火墙通过身份识别，辨别请求访问内部网络者的身份，然后根据该用户所获得的授权，控制其访问授权范围的内容，保护网络的内部信息。防火墙还可以对所提供的网络服务进行控制，通过限制一些不安全的服务，减少威胁，提高网络安全的保护程度。

(2) 内容控制。防火墙可以对穿越防火墙的数据内容进行控制，阻止不安全的数据内容进入内部网络，影响内部网络的安全。病毒、木马等程序经常隐藏在可执行文件或ActiveX控件中，可以通过限制内部人员从外网下载，达到减少威胁的目的。

(3) 安全日志。因所有进出内部网络的通信都必须经过防火墙，故防火墙可以完整地

记录网络通信情况。通过分析、审计日志文件，可以发现潜在的威胁，并及时调整安全策略进行防范；还可以在发生网络破坏事件时，发现破坏者。

(4) 集中管理。防火墙需要针对不同的网络情况与安全需求，制定不同的安全策略，并且还要根据情况的变化改进安全策略。在一个网络的安全防护体系中，会有多台防火墙分布式部署，便于进行集中管理，实施统一的安全策略，避免出现安全漏洞。

(5) 其他附加功能。此外，防火墙还有其他一些附加功能，如支持 VPN(Virtual Private Network，虚拟专用网)、NAT(Network Address Translation，网络地址转换)等。因防火墙所处的位置是网络的出入口，它是支持 VPN 连接的理想接点。目前许多防火墙都提供 VPN 连接功能。而 NAT 技术主要是为了解决 IPv4 的 IP 地址即将耗尽的问题，通过将内部网络地址与外部网络地址进行转换，大大节约对外部网络 IP 地址的使用，减缓耗尽 IP 地址的速度。NAT 相当于网络级的代理，将内部网络计算机的 IP 地址转换成防火墙的 IP 地址，代表内部网络的计算机与外部网络通信，从而使黑客无法获取内部网络计算机的 IP 地址，也就无法有针对性地实施攻击。

6.1.2 防火墙的技术原理

1986 年，Digital 公司在 Internet 上安装了全球第一个商用防火墙系统，之后，相关技术与应用得到了快速的发展，经历了包过滤技术、代理服务技术、状态检测技术等一系列历程。

1. 包过滤技术

包过滤(Packet Filtering)是指防火墙在网络层中(如图 6-2 所示)，通过检查网络数据流中数据包的报头(如源、目的地址、协议类型、端口等)，将报头信息与事先设定的过滤规则进行比较，据此决定是否允许该数据包通过，其关键是过滤规则的设计。包过滤技术是最早应用于防火墙的技术，也是最简单、某些情形下最有效的防火墙技术。

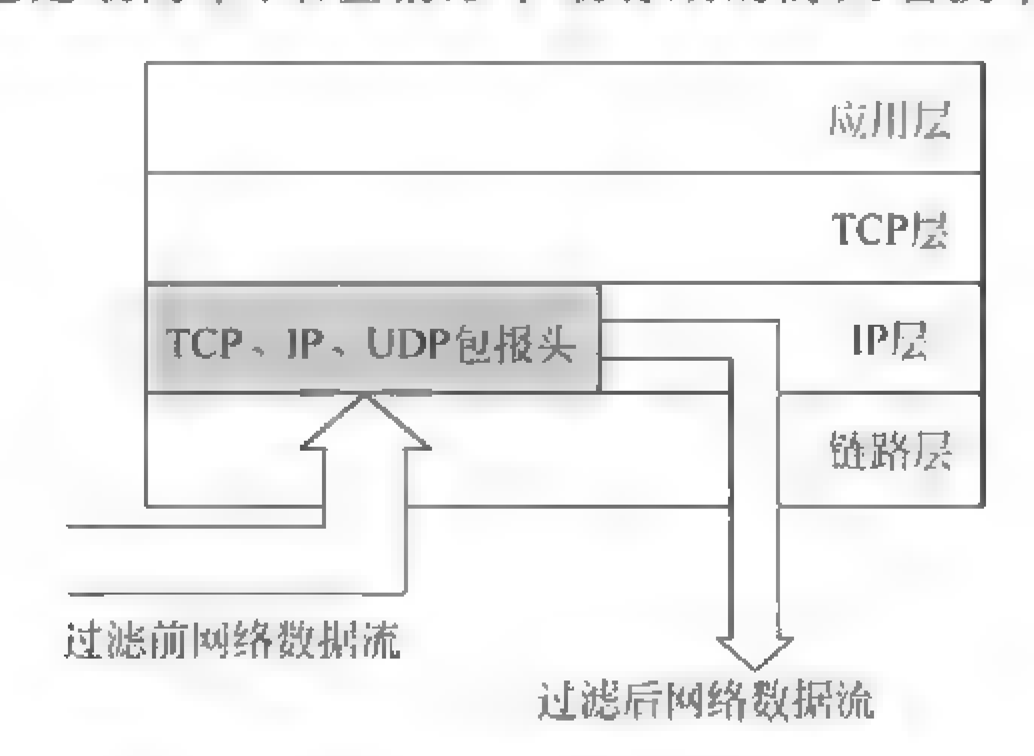


图 6-2　包过滤技术示意图

包过滤技术检查的数据包报头信息主要有以下内容。

(1) IP 数据包的源 IP 地址、目的 IP 地址、协议类型、选项字段等。建立按 IP 地址进行访问控制的安全策略，对 IP 数据包报头信息进行过滤，如果是基于源 IP 地址的过滤规则，只允许特定 IP 地址的外部主机与内部网络连接，拒绝其他主机的连接。如果是基于目的 IP 地址的过滤规则，则外部主机只能访问特定的内部公共服务器。但这种策略对服务器的访问控制太弱，通过增加对协议类型(TCP、UDP、ICMP 等)和端口的过滤规则，来进一步加

强对公共服务器的访问控制。

(2) TCP 数据包的源端口、目标端口、标志段等。端口号为1024以下的TCP端口被用于一些标准的通信服务，如表6-1所示。

表 6-1 一些常用的 TCP 端口

端口	协议	用途
21	FTP	文件传输
23	Telnet	远程登录
25	SMTP	电子邮件
69	TFTP	简单文件传输协议(Trivial FTP)
79	Finger	查询有关一个用户的信息
80	HTTP	WWW 服务
110	POP3	远程电子邮件
119	NNTP	USENET 新闻

如果只允许HTTP通信，而不允许Telnet通信，则通过设定允许TCP端口80的通信、禁止TCP端口23的通信，即可简单方便地对这两项服务进行过滤。

(3) UDP 数据包的源端口、目标端口。UDP的应用有DNS(Domain Name System，域名系统)、RPC(Remote Procedure Call，远程调用)、RTP(Real time Transport Protocol，实时传输协议)等，同样，通过设定基于UDP端口的过滤规则，可以方便地对各项服务进行过滤。

(4) ICMP 类型。ICMP(Internet Control Message Protocol，Internet 控制消息协议)主要用于传递控制或错误消息，如常用的端到端故障查找工具ping就是利用ICMP中的"回应请求"(ICMP类型编号8)实现的。因此，通过设定ICMP关键字或类型编号的过滤规则，就可以对ICMP通信进行过滤，如表6-2所示。

表 6-2 一些常见的 ICMP 类型

ICMP 类型编号	ICMP 类型名称	可能的控制原因
0	回应答复	对 ping 的响应
3	无法到达目的地	无法到达目标地址
4	源端抑制	路由器接收通信量太大
8	回应请求	常规的 ping 请求
11	超时	到目的地时间超时

基于包过滤技术的防火墙具有简单、有效、不需内部网络用户做任何配置、对用户完全透明的优点，但也有不可避免的内在弱点。

(1) 只能检查数据包的报头信息，无法检查数据包的内容，不能进行数据内容级别的访问控制；

(2) 没有考虑数据包的上下文关系，每一个数据包都要与设定的规则匹配，影响数据包的通过速率，无法满足一些访问控制的要求；

(3) 过滤规则的制定很复杂，容易产生冲突或漏洞，出现因配置不当带来的安全问题。

2. 状态检测技术

一个正常网络连接中的源和目的地址、协议类型、协议信息(如 TCP/UDP 端口、ICMP 类型)、标志(如 TCP 连接状态标志)等构成该连接的状态表,将数据包报头的相关信息与状态表进行对比,就可以知道该数据包是一个新的网络连接还是某个已有连接中的数据包。状态检测技术也叫动态包过滤技术,是包过滤技术的延伸。基于状态检测(Stateful Inspection)的防火墙在包过滤防火墙的基础上,增加了对状态的检测,其工作原理如图 6-3 所示。

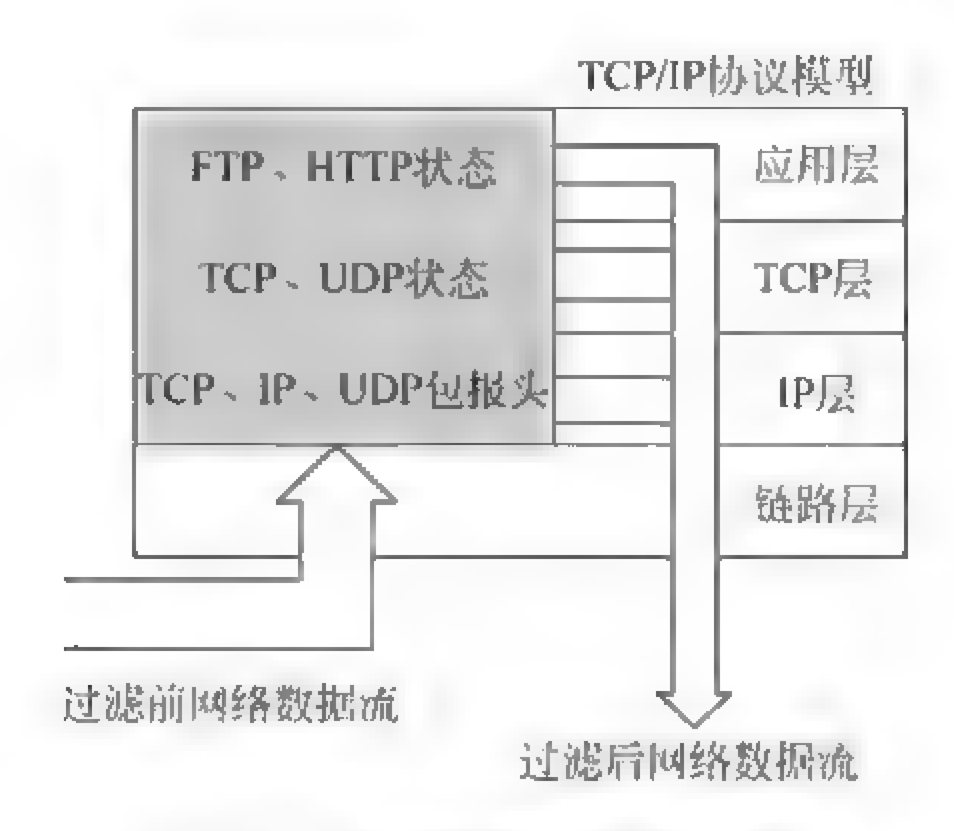


图 6-3 状态检测技术示意图

状态检查的具体流程如下:首先,检测数据包是否是状态表中已有连接的数据包,如果是且状态正确,则允许通过;如果不是,则进行包过滤技术的检查。一旦包过滤允许通过,则在状态表中添加其所在的连接。当某个连接结束或超时,则在状态表中删除该连接信息。

状态检测防火墙的数据包过滤规则是预先设定的,但状态表是动态建立的,可以实现对一些复杂协议建立的临时端口进行有效的管理。如 FTP 只是通过 21 端口进行控制连接,其数据传送是通过动态端口建立的另一个子连接进行传送。如果边界部署的是一个基于包过滤技术的防火墙,就需要将所有端口打开,将会带来很大的安全隐患。但对于基于状态检测技术的防火墙,则能够通过跟踪、分析控制连接中的信息,得知控制连接所协商的数据传送子连接端口,在防火墙上将该端口动态开启,并在连接结束后关闭,保证内部网络的安全。

状态检测技术是为每一个会话连接建立、维护其状态信息,并利用这些状态信息对数据包进行过滤。如状态检测可以很容易地实现只允许一个方向通信的"单向通信规则",在允许通信方向上的一个通信请求被防火墙允许后,将建立该通信的状态表,该连接在另一个方向的回应通信属于同一个连接,因此将被允许通过。这样就不必在过滤规则中为回应通信制定规则,可以大大减少过滤规则的数量和复杂性;而且也不需要对同一个连接的数据包进行检查,从而提高过滤效率和通信速度。

动态状态表是状态检测防火墙的核心,利用其可以实现比包过滤防火墙更强的控制访问能力。但其弱点是也没有对数据包的内容进行检查,不能进行数据内容级别的控制,而且也允许外部主机与内部主机的直接连接,容易遭受黑客的攻击。

3. 代理服务

代理服务(Proxy Server)是代表内部网络与外部网络进行通信的服务器,通信发起方

首先与代理服务建立连接,然后代理服务再另外建立到目标主机的连接,通信双方通过代理进行间接连接、通信,不允许端到端的直接连接。各种网络应用服务也是通过代理提供,由此达到访问控制的目的。

1) 应用级代理

应用级代理也被称为应用级网关(Application Gateway),工作在应用层,是一组特殊的应用服务程序,其工作原理如下。

(1) 当接收到客户方发出的连接请求后,应用代理检查客户的源和目的IP地址,并依据事先设定的过滤规则决定是否允许该连接请求。

(2) 如果允许该连接请求,进行客户身份识别。否则,则阻断该连接请求。

(3) 通过身份识别后,应用代理建立该连接请求的连接,并根据过滤规则传递和过滤该连接之间的通信数据。

当一方关闭连接后,应用代理关闭对应的另一方连接,并将这次的连接记录在日志内。

应用代理服务器一般运行在具有两个网络接口的双重宿主主机的防火墙上,两个网络接口分别连接内、外网络,并且禁止IP转发,切断内外网络之间直接的IP通信,由代理服务器按照一定的安全策略提供Internet连接和服务。

以电子邮件应用代理为例。正常的电子邮件传输是发起方与接收方首先通过建立邮件传输合法性的协议连接,然后传输邮件。加入电子邮件应用代理后,发起方与接收方通过代理在中间做转发通信,代理既代表发起方与接收方通信,也代表接收方与发起方通信。代理工作在应用层,可以对数据内容进行审查,对垃圾邮件等含有不良信息的邮件进行过滤。

同前两种防火墙技术相比,代理服务器技术的优点如下。

(1) 内部网络的拓扑、IP地址等被代理防火墙屏蔽,能有效实现内外网络的隔离。

(2) 具有强鉴别和日志能力,支持用户身份识别,实现用户级的安全。

(3) 能进行数据内容的检查,实现基于内容的过滤,对通信进行严密的监控。

(4) 过滤规则比数据包过滤规则简单。

其缺点如下。

(1) 代理服务的额外处理请求降低了过滤性能,其过滤速度比包过滤器速度慢。

(2) 需要为每一种应用服务编写代理软件模块,提供的服务数目有限。

(3) 对操作系统的依赖程度高,容易因操作系统和应用软件的缺陷而受到攻击。

2) 电路级代理

电路级代理也被称为电路级网关,是一个通用代理服务器,工作在传输层(TCP层),可以认为是包过滤技术的延伸,但它不像包过滤技术那样只是基于IP地址、端口号等报头信息进行过滤,还能进行用户身份鉴别。而且对于已经建立连接的网络数据包,电路级代理不再对其进行过滤。

与应用级代理相比较,电路级代理不用为不同的应用开发不同的代理模块,具有较好的通用性。但因也对网络数据包进行了复制、转发,因此具有占用资源大、速度慢的缺点,而且包过滤技术的缺点在这里也同样存在。

6.1.3 防火墙的应用

在介绍防火墙的体系结构之前,先介绍一下堡垒主机的概念。顾名思义,堡垒主机就是

位于内部网络的最外层，像堡垒一样防护内部网络的设备。堡垒主机是防火墙体系结构中暴露在Internet上、最容易遭受攻击的设备，因此对其安全性要给予特别的关注。在实际应用中，防火墙技术的应用不是单一的，而是结合了多种技术来构筑防火墙的体系结构，实现一个实用、有效的防火墙系统。

防火墙体系结构如下。

(1) 屏蔽路由器结构。这是一种最简单的体系结构，屏蔽路由器(或主机)作为内外连接的唯一通道，对进出网络的数据进行包过滤，其结构如图6-4所示。

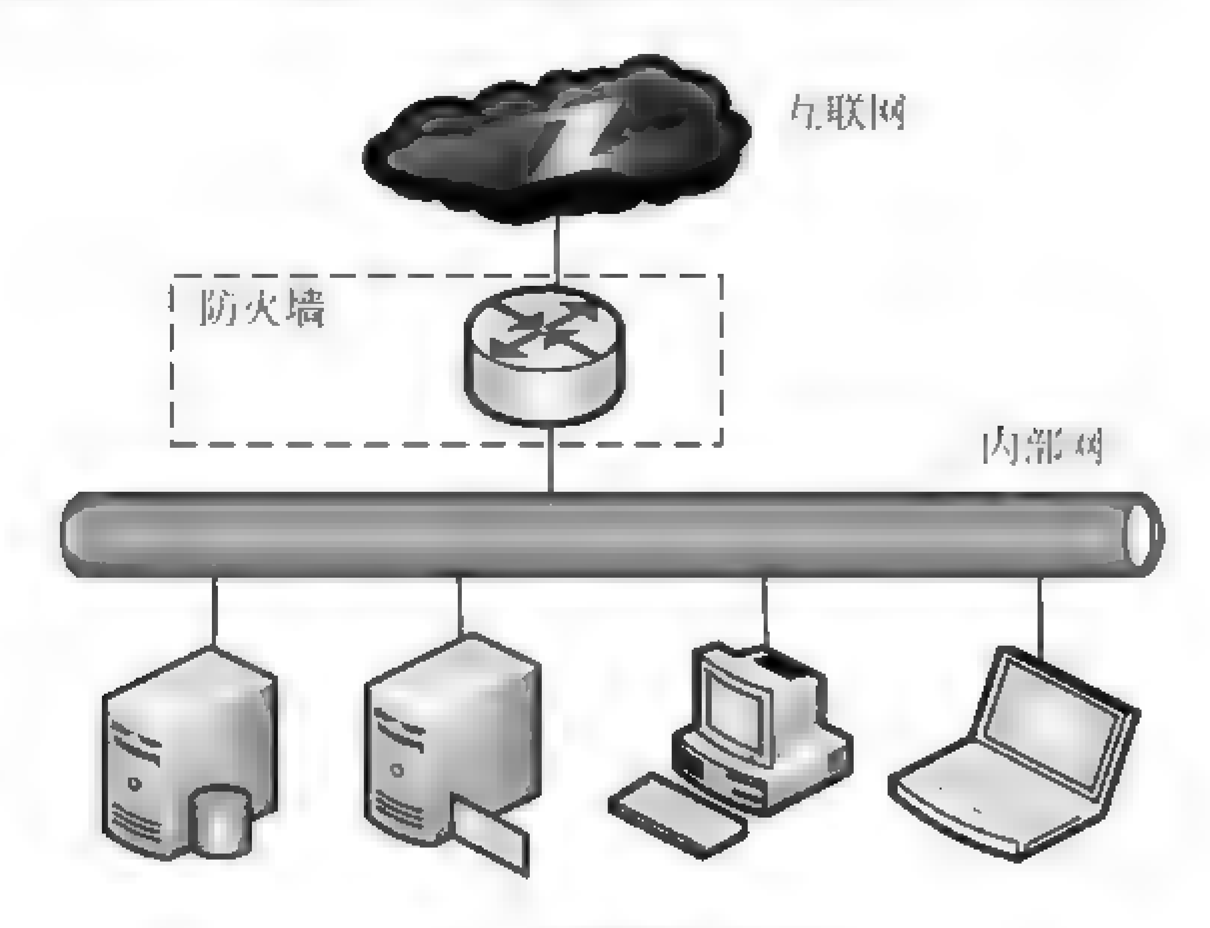


图6-4　屏蔽路由结构

(2) 双重宿主主机结构。双重宿主主机是至少有两个网络接口的主机，一个网络接口连接内部网络，另一个网络接口连接外部网络，因此主机可以充当内外网的路由器，并能从一个网络向另一个网络直接发送IP数据包，其结构如图6-5所示。

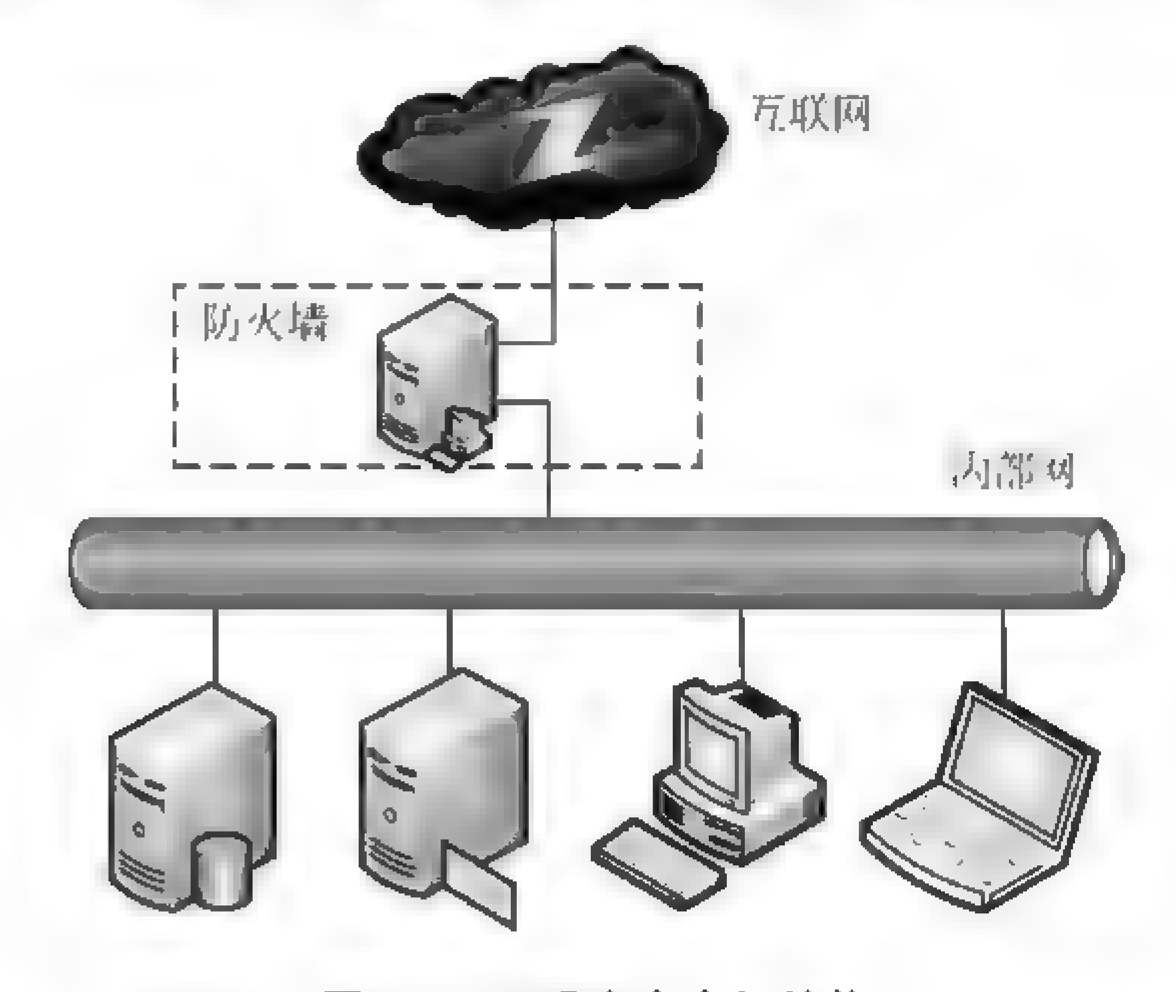


图6-5　双重宿主主机结构

内部网络与外部网络通过双重宿主主机的过滤、转接方式进行通信，而不是直接的IP通信，双重宿主主机为不同的服务提供代理。双重宿主主机充当了堡垒主机的角色，其弱点就在于主机的脆弱性，一旦入侵者攻破堡垒主机，使其仅仅为一个路由器，则外部网络的用户就可以直接访问内部网络。

(3) 屏蔽主机结构。屏蔽主机结构的防火墙使用一个路由器隔离内部网络和外部网络,代理服务器堡垒主机部署在内部网络上,并在路由器上设置数据包过滤规则,使堡垒主机成为外部网络唯一可以访问的主机,通过路由器的包过滤技术和堡垒主机的代理服务器技术防护内部网络的安全,如图6-6所示。屏蔽主机的防火墙体系结构易于实现,而且比双重宿主主机结构的安全性高,应用比较广泛。

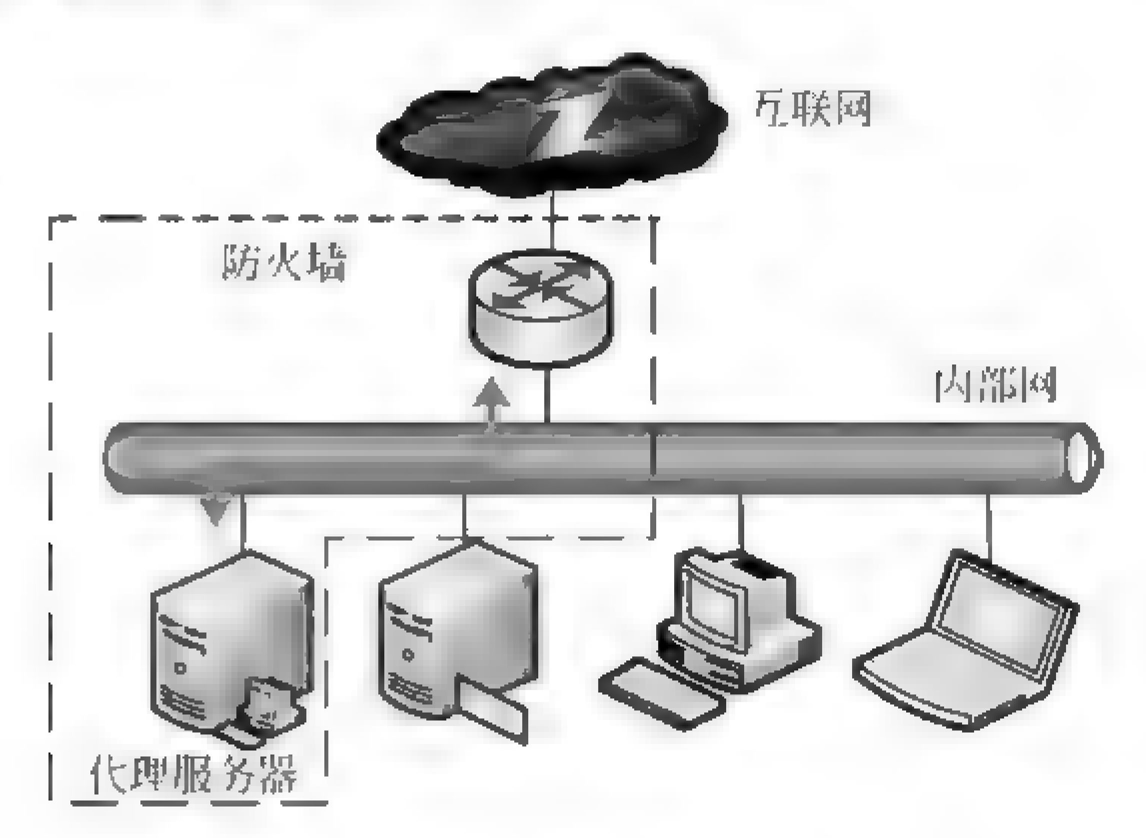


图 6-6 屏蔽主机结构

(4) 屏蔽子网结构。屏蔽子网结构的防火墙通过建立一个周边网络来分隔内部网络和外部网络,进一步提高防火墙的安全性。其结构如图6-7所示。

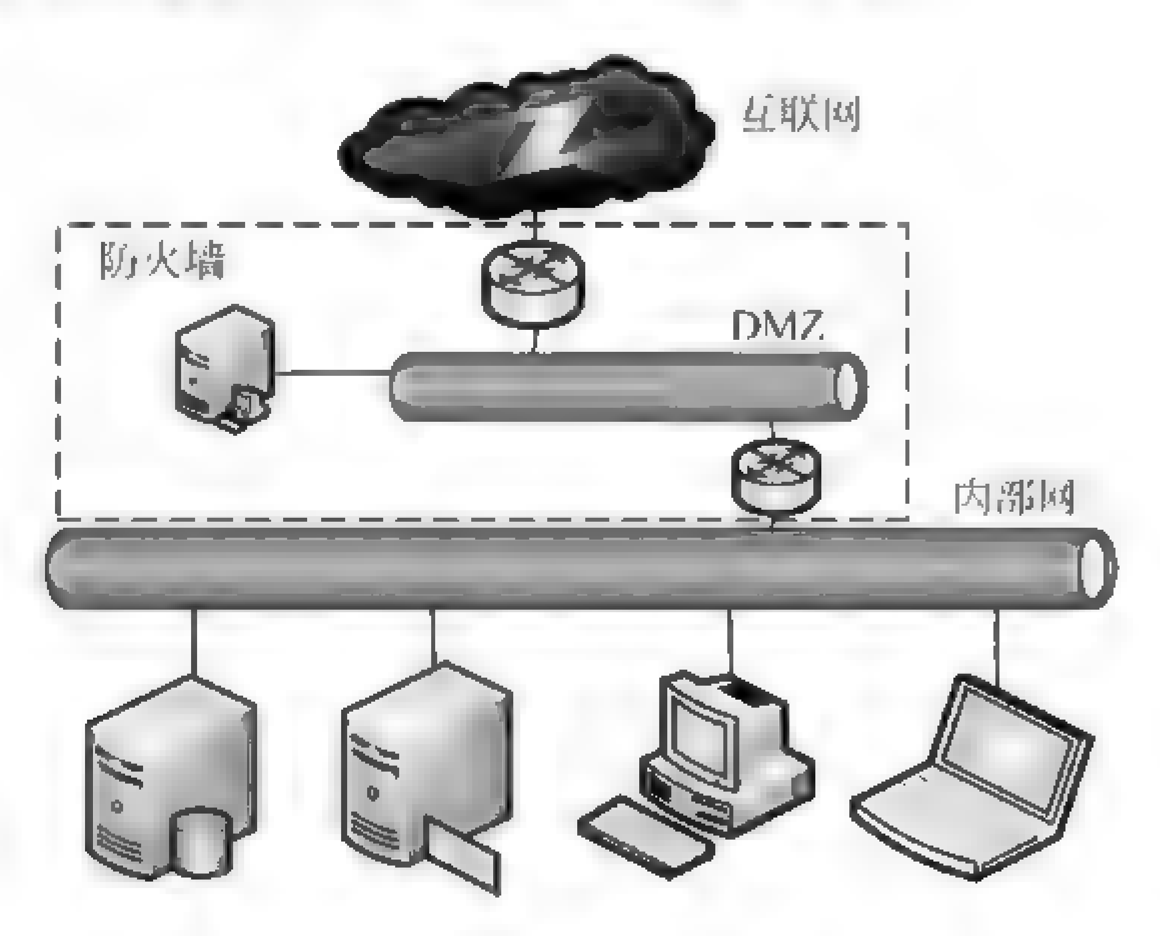


图 6-7 屏蔽子网结构

周边网络是一个被隔离的子网,在内、外之间形成一个"非军事化区"(De Militarized Zone,DMZ)的隔离带。这种结构的防火墙最简单的形式是用两个屏蔽路由器把周边网络分别与内部网络、外部网络分开,一个路由器控制外部网络数据流,另一个路由器控制内部网络数据流,内部网络和外部网络均可访问周边网络,但不允许穿过周边网络进行通信。在屏蔽子网结构中还可以根据需要在屏蔽子网中安装堡垒主机,为内部网络与外部网络之间的通信提供代理服务,但对堡垒主机的访问都必须通过两个屏蔽路由器。

如果攻击者试图完全破坏屏蔽子网结构的防火墙,则需要重新配置连接外部网络、周边网络、内部网络的路由器,大大增加了攻击的难度。如果进一步禁止访问路由器或者只允许

内部网络中的特定主机才可以访问，则攻击会变得更加困难。屏蔽子网结构的防火墙具有很高的安全性，但所需设备较多，费用较高，而且实施和管理都比较复杂。

6.1.4 防火墙的局限性

作为内部网络与外部公共网络之间的第一道屏障，防火墙是最先受到人们重视的网络安全产品之一。从理论上看，防火墙处于网络安全的最底层，负责网络间的安全认证与传输，但随着网络安全技术的整体发展和网络应用的不断变化，现代防火墙技术已经逐步走向网络层之外的其他安全层次，不仅要完成传统防火墙的过滤任务，同时还能为各种网络应用提供相应的安全服务。除此之外，还有多种防火墙产品正朝着数据安全与用户认证、防止病毒与黑客侵入等方向发展。

虽然防火墙能在网络边界对受保护网络进行很好的保护，但是并不能解决所有的安全问题。首先，防火墙只是一种边界安全保护系统，要保证边界的所有出口都有防火墙的保护，才能形成对网络边界内环境的防护。其次，防火墙只能保护边界内的环境，通信数据在穿越边界出去后，将失去防火墙的防护。而且内部人员发起的攻击，因没有经过防火墙，所以防火墙也无法提供防护。最后防火墙的配置是基于已知攻击知识制定的，因此无法对新的攻击方式进行防护，需要经常更新配置。另外，防火墙对通信内容的控制很弱，因此其对病毒、蠕虫、木马等恶意代码的防护能力不尽人意。

因此，不能认为安装了防火墙，内部网络的安全问题就可以彻底解决了，需要结合其他安全技术，构建不同层次、不同深度的防御体系。

6.2 实例编程——实现包过滤防火墙

包过滤防火墙是最原始的防火墙，现在的绝大多数路由器都具有包过滤功能，因此路由器可以作为包过滤防火墙。使用包过滤防火墙前，要制定规则，这些规则说明什么样的数据能够通过，什么样的数据禁止通过，多条规则组成一个访问控制列表(Access Control List，ACL)。对所有数据，防火墙都要检查它与ACL的规则是否匹配。在确定过滤规则之前，需要做如下决定。

(1) 打算提供何种网络服务，并以何种方向(从内部网络到外部网络，或者从外部网络到内部网络)提供这些服务。

(2) 是否限制内部主机与因特网进行连接。

(3) 因特网上是否存在某些可信任主机，它们需要以什么形式访问内部网。

包过滤防火墙根据每个包头部的信息来决定是否要将包继续传输，从而增强安全性。对于不同的包过滤防火墙，用来生成规则进行过滤的包头部信息不完全相同，但通常都包括以下信息。

(1) 接口和方向：包是流入还是离开网络，这些包通过哪种接口。

(2) 源和目的IP地址：检查包从何而来(源IP地址)、发往何处(目的IP地址)。

(3) IP选项：检查所有选项字段，特别是要阻止源路由(Source Routing)选项。

(4) 高层协议：使用IP包的上层协议类型，例如TCP还是UDP。

(5) TCP 包的 ACK 位检查：这一字段可帮助确定是否有，及以何种方式建立连接。

(6) ICMP 的报文类型：可以阻止某些刺探网络信息的企图。

(7) TCP 和 UDP 包的源端口和目的端口：此信息帮助确定正在使用的是哪些服务。

创建包过滤防火墙的过滤规则时，要注意以下重要事项。

(1) 在规则中要使用 IP 地址，而不要使用主机名或域名。虽然进行 IP 地址欺骗或域名欺骗都不是非常难的事，但在很多攻击中，IP 地址欺骗常常是不容易做到的，因为黑客想要真正得到响应并非易事。然而只要黑客能够访问 DNS 数据库，进行域名欺骗却是很容易的事。这时，域名看起来是真实的，但它对应的 IP 地址却是另一个虚假的地址。

(2) 不要回应所有从外部网络接口来的 ICMP 数据，因为它们很可能给黑客暴露信息，特别是哪种包可以流入网络，哪种包不可以流入网络的信息。响应对某些 ICMP 数据可能等于告诉黑客，在某个地方确实有一个包过滤防火墙在工作。在这种情况下，对黑客来说有信息总比没有好。防火墙的主要功能之一就是隐藏内部网络的信息，黑客通过对信息的筛选处理，可以发现什么服务不在运行，而最终发现什么服务在运行。如果不响应 ICMP 数据，就可以限制黑客得到可用的信息。

(3) 要丢弃所有从外部进入，而其源 IP 地址是内部网络的包。这很可能是有人试图利用这些包进行 IP 地址欺骗，以达到通过网络安全关口的目的。

(4) 防火墙顺序使用 ACL 中的规则，只要有一条规则匹配，就采取规则中规定的动作，后面的规则不再使用。所以规则的顺序非常重要，错误的顺序可能使网络不能正常工作，或可能导致严重的安全问题。

6.2.1 基于协议的数据包过滤实现

该过程要实现的功能是根据传输层协议来进行数据包的过滤。即拒绝 ICMP 包，只允许 TCP 包和 UDP 包通过。

程序中的钩子函数定义如下。

```
/* definition of hook function */
unsigned int hook_func(unsigned int hooknum,
                       struct sk_buff ** skb,
                       const struct net_device * in,
                       const struct net_device * out,
                       int ( * okfn)(struct sk_buff * ))
{
    struct sk_buff * pskb = * skb;
    switch(pskb->nh.iph->protocol)
    {
    case IPPROTO_ICMP:
        {
        printk("ICMP Packet: DROP\n");
        return NF_DROP;
        }
    case IPPROTO_TCP:
        {
        printk("TCP Packet: ACCEPT\n");
        return NF_ACCEPT;
```

```
        }
    case IPPROTO_UDP:
        {
        printk("UDP Packet: ACCEPT\n");
        return NF_ACCEPT;
        }
    default:
        {
        printk("Unknown Packet: DROP\n");
        return NF_DROP;
        }
    }
}
```

程序可以通过一个 sk_buff 指针来定位数据包的 IP 头部，然后根据该头部的协议字段进行数据报的过滤。

在内核编程中，不能使用用户态 C 语言库函数中的 printf( )函数来输出信息，而只能使用 printk( )函数，但尽管使用 printk( )函数来输出信息，在控制台也不能看到输出的信息，这是因为内核中 printk( )函数的设计目的并不是为了和用户交流，它实际上是内核的一种日志机制，是用来记录日志信息或给出警示提示的。每个 printk( )函数都会有一个优先级，内核一共有 8 个优先级，它们都有对应的宏定义。如果未指定优先级，内核会选择默认的优先级 DEFAULT_MESSAGE_LOGLEVEL。如果 printk 的优先级比当前终端的优先级等级高，消息就会打印到控制台上。因此如果想要在控制台上看到 printk( )函数的输出信息，可以设置一下 printk( )函数的优先级，使其比当前终端的优先级等级高，就可以向终端上输出信息了。

6.2.2　基于源 IP 地址的数据包过滤实现

该过程要实现的功能是对源 IP 地址为"192.168.1.27"的数据报全部丢弃。程序的钩子函数定义如下。

```
/* definition of hook function */
unsigned int hook_func(unsigned int hooknum,
                       struct sk_buff **skb,
                       const struct net_device *in,
                       const struct net_device *out,
                       int (*okfn)(struct sk_buff *))
{
    struct sk_buff *pskb = *skb;
    if((pskb->nh.iph->saddr) == in_aton("192.168.1.27"))
    {
     printk("<0>""A Packet from 192.168.1.27: DROP\n");
     return NF_DROP;
    }
    else
    {
     return NF_ACCEPT;
    }
}
```

6.2.3 基于 TCP 通信目的端口过滤实现

该过程要实现的功能是对于目的端口号为 23 的 TCP 包全部丢弃。程序的钩子函数定义如下。

```
/* definition of hook function */
unsigned int hook_func(unsigned int hooknum,
                              struct sk_buff ** skb,
                              const struct net_device * in,
                              const struct net_device * out,
                              int ( * okfn)(struct sk_buff * ))
{
    struct sk_buff * pskb = * skb;
    struct tcphdr * thdr = (struct tcphdr * )(pskb->data + (pskb->nh.iph->ihl * 4));
    if((pskb->nh.iph->protocol)!= IPPROTO_TCP)
    {
    printk("<0>""Not A TCP Packet: ACCEPT\n");
    return NF_ACCEPT;
    }
    else
    {
    if(thdr->dest == in_pton("23"))
    {
        printk("<0>""A TCP Packet PORT 23: DROP\n");
        return NF_DROP;
    }
    else
        return NF_ACCEPT;
    }
}
```

6.2.4 包过滤防火墙的编程实现

包过滤防火墙的代码实现如下。

```
//xieyi.h
#include<iostream>
using namespace std;
intxieyi()
        {
            cout<<"请输入要拒绝的数据包协议的名字。如:icmp、tcp、udp(小写字母)"<<endl;
            string a;
            inti;
            cin>>a;
            if(a == "icmp")
            {
                cout<<"ICMP Packet:DROP"<<endl;
                return 0;
```

```
            }
            else
                if(a == "tcp")
                {
                    cout <<"TCP Packet:DROP"<< endl;
                    return 0;
                }
                else
                    if(a == "udp")
                    {
                        cout <<"UDP Packet:DROP"<< endl;
                    return 0;
                    }
                    else
                    {
                        cout <<"Unknown Packet:DROP"<< endl;
                        return 0;

                    }
      }

//ip.h
#include< iostream >
using namespace std;
intip()
{
            cout <<"请输入一个 IP 地址:(本次禁止的 IP 地址为 192.168.1.27)"<< endl;
            stringip;
            cin >> ip;
            if(ip == "192.168.1.27")
            {
                cout <<"A Packet from 192.168.1.27.DROP"<< endl;
                return 0;
            }
            else
            {
                cout <<"A Packet from "<< ip <<":ACCEPT"<< endl;
                return 1;
            }

        }

//duankou.h
#include< iostream >
using namespace std;
intduankou()
    {
            cout <<"该程序实现的是对于目的端口为"23"的 TCP 包全部丢弃"<< endl;
            string a;
            cout <<"请输入协议类型:(小写字母)"<< endl;
```

```
            cin>>a;
            if(a!="tcp")
            {
                cout<<"NOT A TCP Packet:ACCEPT"<<endl;
                return 1;
            }
            else
            {
                int s;
                cout<<"请输入端口号:"<<endl;
                cin>>s;
                if(s==23)
                {
                    cout<<"A TCP Pactet PORT 23:DROP"<<endl;
                    return 0;
                }
                else
                {
                    cout<<"NOT A TCP Pactet PORT 23:ACCEPT"<<endl;
                    return 1;
                }
            }

        }
//text.cpp

#include<iostream>
#include<string>
#include"xieyi.h"
#include"duankou.h"
#include"ip.h"

using namespace std;
int main()
{
cout<<"****************************************"<<endl;
cout<<"*          欢迎使用基于包过滤技术防火墙          *"<<endl;
cout<<"*              请选择需要的过滤方法              *"<<endl;
cout<<"*              1.基于协议过滤                    *"<<endl;
cout<<"*              2.基于源IP地址过滤                *"<<endl;
cout<<"*              3.基于TCP通信目的端口过滤         *"<<endl;
cout<<"*              0.退出                            *"<<endl;
cout<<"****************************************"<<endl;

int c;
cin>>c;
while(c)
{
switch(c)
{
```

```
case 1:
    xieyi();
    break;
case 2:
    ip();
    break;
case 3:
    duankou();
    break;
default:
    cout <<"输入错误: "<< endl;
    break;
}
cout <<"********** 请选择需要的过滤方法 *********"<< endl;
cin >> c;
}
}
```

运行结果如图 6-8 所示。



图 6-8　包过滤防火墙运行结果图

小　　结

防火墙提供了网络之间或网络对主机的访问控制，以及地址隐藏等技术手段，保护网络资源免受非法侵害，是目前常用的网络安全设备之一。本章针对防火墙技术展开了详细的介绍，围绕着防火墙的基本工作原理、防火墙的实现方式、常见的防火墙类型展开了阐述，并用实例来展示了如何实现简单的防火墙功能，实现了针对基于协议的数据报过滤功能、基于源 IP 地址的数据报过滤功能、基于目的端口的 TCP 包过滤功能，在此基础上，实现包过滤防火墙。

思 考 题

1. 什么是防火墙?

2. 阐述防火墙的体系结构,分析比较其优缺点。

3. 设计并实现基于目的 IP 地址的数据报过滤功能。

4. 设计并实现基于源端口的 UDP 包过滤功能。

5. 设计并实现一个防火墙,要求所有来自 192.168.1.0~192.168.1.254 网段的数据报都设置为接受;对于 202.113.25.0~202.113.25.254 网段的 IP 数据报,只允许来自 202.113.25.174 的数据报通过,其余的丢掉;拒绝其他主机通过 Telnet 连接本机,允许通过 FTP 连接本机。

6. 除了包过滤技术外,实现防火墙的安全技术还有哪些?请尝试实现其中一种应用技术。

第 7 章　入侵检测模型设计与实现

如果攻击者成功地绕过防御措施，渗透到网络中，如何检测出攻击行为呢？而且，内部人员所发送的攻击防御措施也是无济于事的。入侵检测技术属于事后检测技术，对于保护网络资源也是重要手段之一。本章将介绍入侵检测技术的基本工作原理，研究入侵检测系统的设计与软件编程的方法。

7.1　入侵检测技术

利用防火墙技术，经过仔细配置，通常能够在内外网之间提供安全的网络保护，降低了网络安全风险。但是，仅使用防火墙，网络安全还远远不够。

(1) 入侵者可寻找防火墙背后可能敞开的后门。

(2) 入侵者可能就在防火墙内。

(3) 由于性能限制，防火墙通常不能提供实时的入侵检测能力。

入侵检测系统(Intrusion Detection System，IDS)通过监视受保护系统或网络的状态和活动，发现正在进行或已发生的攻击，起到信息保障体系结构中检测的作用。

因此，入侵检测技术的目的是提供实时的入侵检测及采取相应的防护手段，如记录证据用于跟踪和恢复、断开网络连接等。

7.1.1　入侵检测的基本原理

1980 年，J. Anderson 在他的那篇被誉为入侵检测的开山之作的文章 *Computer Security Threat Monitoring and Surveillance* 中首次提出了创建安全审计记录和在此基础上的计算机威胁监控系统的基本构想。首先定义成功的攻击为渗透，为了创建安全审计记录，他对入侵威胁进行了分类，见图 7-1，指出来自内部的渗透者是系统安全的主要隐患，按照检测难度递增，把攻击分为假冒者(假冒他人的内部用户)、误用者(合法用户误用了对系统或数据的访问)、秘密用户(获取了对系统的管理控制)。至于来自外部的渗透者，当他们成功地突破了目标系统的访问控制后，相应的威胁就转变为内部的威胁。

1. 三类内部渗透者与入侵检测的分析模型

假冒者盗用他人账户信息。他对系统的访问可以看成是对系统的“额外”使用，直觉上，他对系统的访问行为轮廓应该和他所冒充的用户有所不同，因此一个自然的检测方法是在审计记录中为系统的每个合法用户建立一个正常行为轮廓，当检测系统发现当前用户的行为和他的正常行为轮廓有较大偏差时，就应该及时提醒系统安全管理员。这样的检测方法称为异常检测。

误用者是合法用户对系统或数据的越权访问。与授权用户的行为相比，这些越权举动可能在统计上没有显著的区别，因此通过比较当前行为和正常行为轮廓以发现可能的入侵

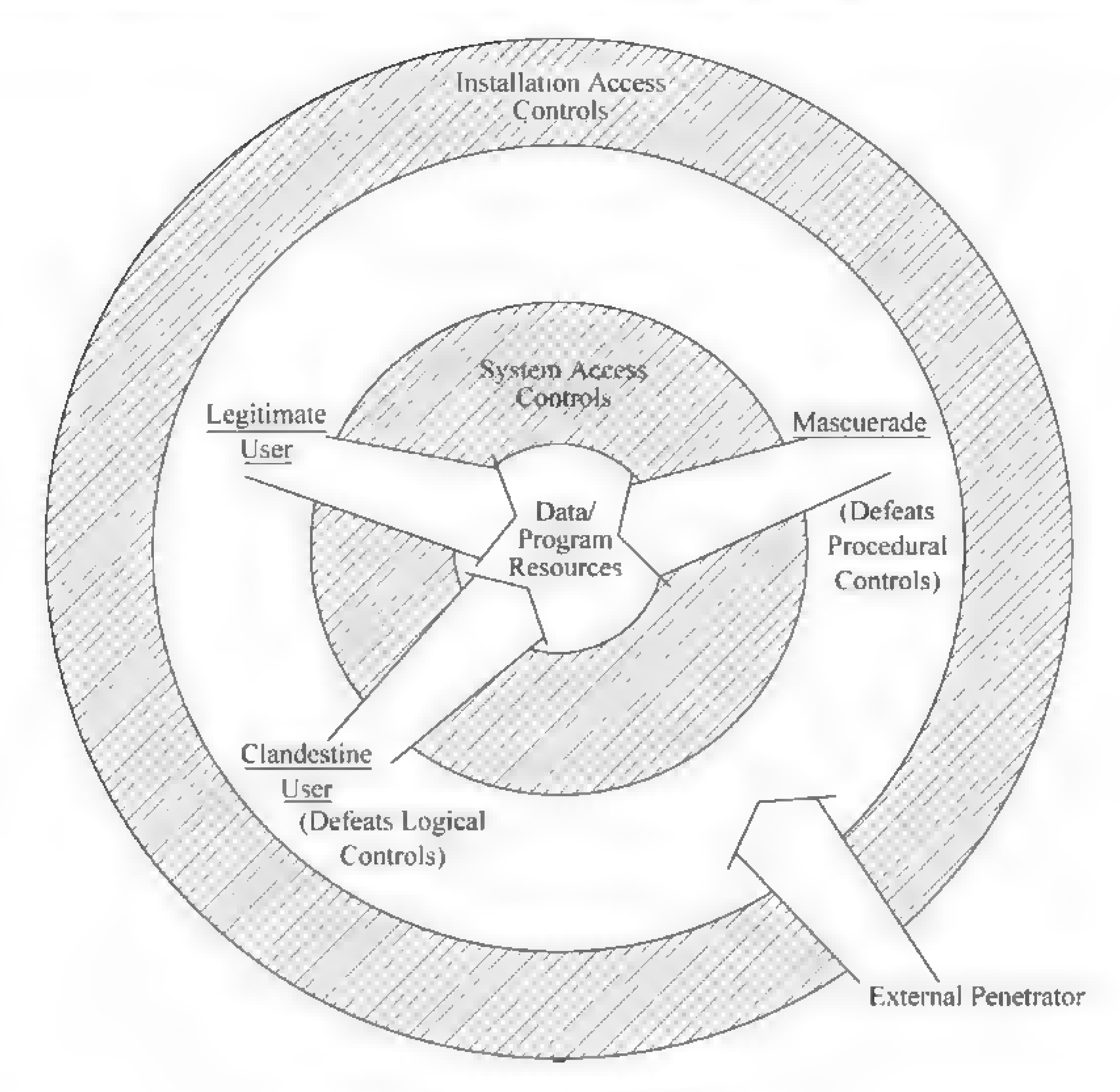


图 7-1 入侵威胁分类

行为的做法，要比假冒者情景困难。然而，如果这些越权举动构成明显的入侵行为，则可以通过事先刻画已知攻击的特征，将越权举动和这些特征相匹配，从而检测出攻击。这种方法称为误用检测。

秘密用户拥有对系统的管理控制权。可以利用他的权限来躲避审计记录，因此是很难通过安全审计记录来检测出所发生的攻击的，除非他的秘密行动显示出上述两类攻击者的特征。

综上所述，异常检测和误用检测是入侵检测的两种主要分析模型，其中用户正常行为轮廓的建立主要是基于统计的方法，而攻击特征的刻画主要是基于规则。对于假冒者偏向于采用异常检测的方法，对于有不当行为的合法用户偏向于采用误用检测的方法，但在实践中往往两种方法混合使用。

2. 入侵检测的数据源

入侵检测的数据源，是反映受保护系统运行状态的记录和动态数据。最初主要是基于主机的，但从 20 世纪 90 年代开始，网络数据逐渐成为商用入侵检测系统最为通用的数据源，相应的两类入侵检测系统分别称为基于主机和基于网络的入侵检测系统。

基于主机的数据源主要包括：

(1) 操作系统审计记录 — 由专门的操作系统机制产生的系统事件的记录；

(2) 系统日志——由系统程序产生的用于记录系统或应用程序事件的文件。

操作系统的审计记录是系统活动的信息集合，它按照时间顺序组成数个审计文件，每个文件由审计记录组成，每条记录描述了一次单独的系统事件，由若干个域（又称审计标记）组成。当系统中的用户采取动作或调用进程时，引起的系统调用或命令执行，此时审计系统就会产生对应的审计记录。大多数商用操作系统的审计记录是按照可信产品评估程序的标准设计和开发的，具有低层次和细节化的特征，因此成为基于主机的入侵检测系统首选数据源。

系统日志是反映系统事件和设置的文件。例如，UNIX 提供通用的服务 syslog（用于支持产生和更新事件日志）；Sun Solaris 中的 lastlog（记录用户最近的登录，成功或不成功）、pacct（记录用户执行的命令和资源使用的情况）。和操作系统的审计记录相比，系统日志存在如下安全隐患：产生系统日志的软件通常作为应用程序而不是操作系统的子程序运行，易于遭到恶意的破坏和修改；系统日志通常存储在系统未经保护的目录中，而且以文本的形式存储，而审计记录则经过加密和校验处理，为防止篡改提供了保护机制。

但另一方面，系统日志和审计记录相比，具有较强的可读性；而在某些特殊的环境下，可能无法获得操作系统的审计记录或不能对审计记录进行正确的解释，此时系统日志就成为系统安全管理必不可少的信息来源。

网络数据是当前商用入侵检测系统最为通用的数据来源。当网络数据流在检测系统所保护的网段中传播时，采用特殊的数据提取技术，收集网段中传播的数据，作为检测系统的数据来源。和基于主机的数据源相比，它具有如下突出的优势：网络数据是通过网络监听的方式获得的，由于网络嗅探器所做的工作仅仅是从网络中读取传输的数据包，因此对被保护系统的性能影响很小，而且无须改变原有的系统和网络结构；网络监视器与受保护主机的操作系统无关。相比之下，基于主机的入侵检测系统必须针对不同的操作系统开发相应的版本。

3. 入侵检测系统的一般框架

入侵检测系统的一般框架如图 7-2 所示，其中各部分功能介绍如下。

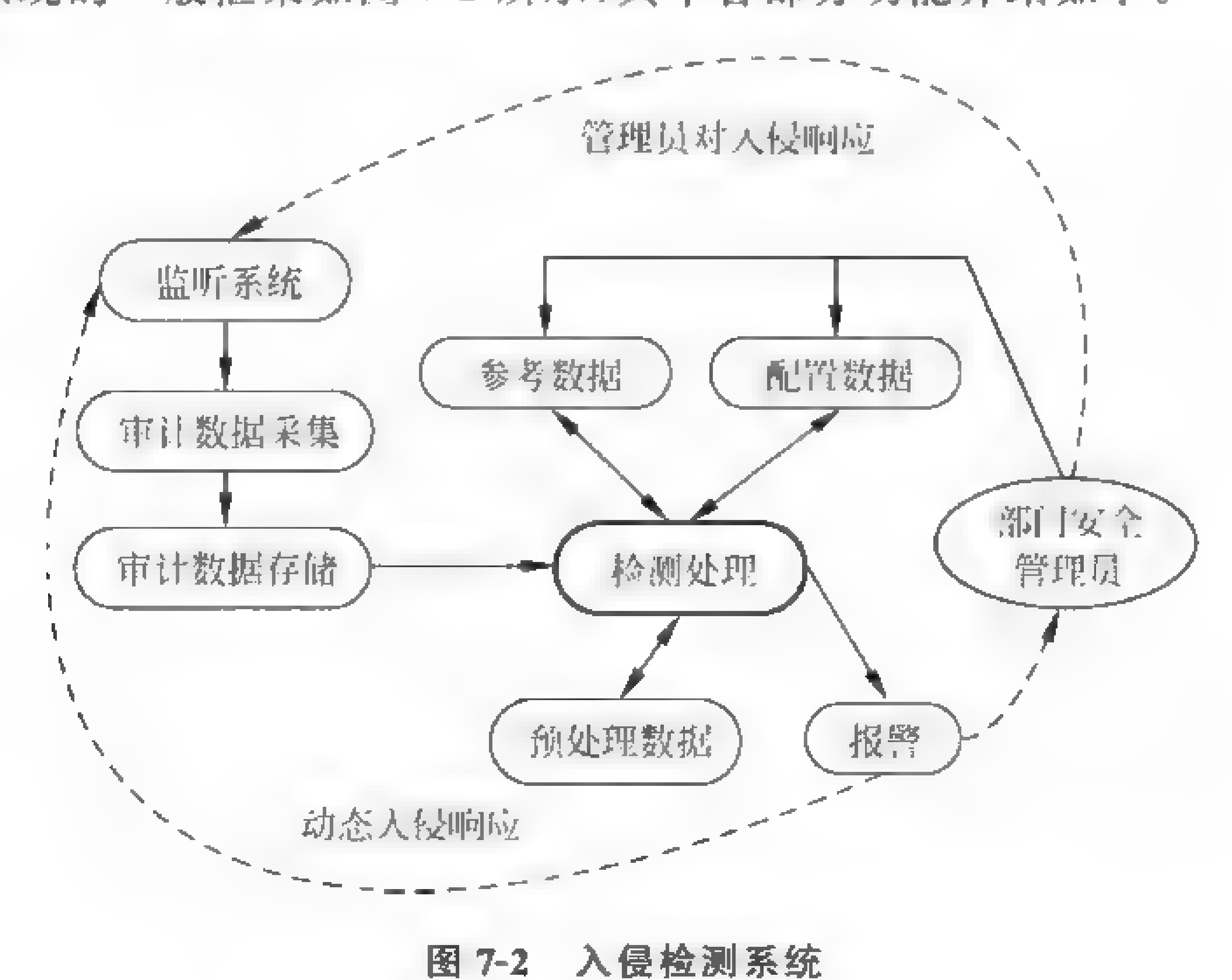


图 7-2　入侵检测系统

（1）审计数据采集：数据源主要是前面所讨论的基于主机和基于网络两个来源。

（2）数据处理（检测）：主要的检测模型是前文所介绍的误用检测和异常检测，它们所采用的主要分析方法分别是基于规则和基于统计。在应用这些方法之前，常常对审计数据进行预处理。

（3）参考数据：主要包括已知攻击的特征和用户正常行为的轮廓，而检测引擎会不断地更新这些数据。

（4）报警：该模块处理由整个系统产生的所有输出，结果可以是对怀疑行动的自动响应，但最为普遍的是通知系统安全管理员。

（5）配置数据：主要指影响检测系统操作的状态，例如，审计数据的来源和收集方法，如何响应入侵等。系统安全管理员是通过配置数据来控制入侵检测系统的运行的。

（6）审计数据存储与预处理：为后期数据处理提供方便的数据检索和状态保存而设置的，可以看成数据处理的一部分。

7.1.2 入侵检测的主要分析模型和方法

1. 异常检测

异常检测最初是基于这样的假设：不同用户之间的正常行为轮廓是可以区分开来的，如用户的计算机登录事件、使用频率等。后来这种假设又推广到特权程序（如 UNIX 中的 setuid 根程序）的预期行为，但无论是用户还是特权程序的行为，异常检测主要由两个步骤组成：①建立正常行为轮廓；②比较当前行为和正常行为轮廓，从而估计当前行为偏离正常行为的程度。异常检测所使用的分析方法也由最初的统计方法拓展到后来的机器学习方法上来。下面将介绍这些建模和分析方法。

刻画用户的正常行为轮廓是建立在 Denning 的工作基础上的。Denning 首次提出一个实时入侵检测专家系统的模型，并根据该模型开发出世界上第一个入侵检测系统原型——IDES（Intrusion Detection Expert System）。该模型由以下 6 个部分组成。

（1）主体：行为的发起者，通常为终端用户。

（2）客体：由系统管理的资源，如文件、命令、装置等。

（3）审计记录：目标系统生成的，对主体在客体上执行或尝试的动作的反映，这些动作包括用户登录、命令执行、文件访问等。

（4）行为轮廓：刻画主体对客体行为的结构，这些结构由观察到的行为的统计度量和模型所描述。

（5）异常记录：观察到异常行为时产生。

（6）动作规则：当某些条件满足时采取的动作，包括更新轮廓、检测异常行为、把异常和怀疑的入侵相关联，以及生成报告。

这 6 个部分构成了一个入侵检测系统，见图 7-3。

整个模型可以看成是一个基于规则的模式匹配系统。每当新生成一个审计记录时，它就和轮廓进行匹配，相匹配轮廓的类型信息决定了应用哪些规则来更新轮廓、检查异常行为和报告所检测到的异常行为。安全管理员帮助建立所要监测的活动的轮廓模板，但规则和轮廓的结构在很大程度上是与系统无关的。

在轮廓部分 Denning 使用三种统计度量：事件计数器、区间计数器（指两个相关事件之

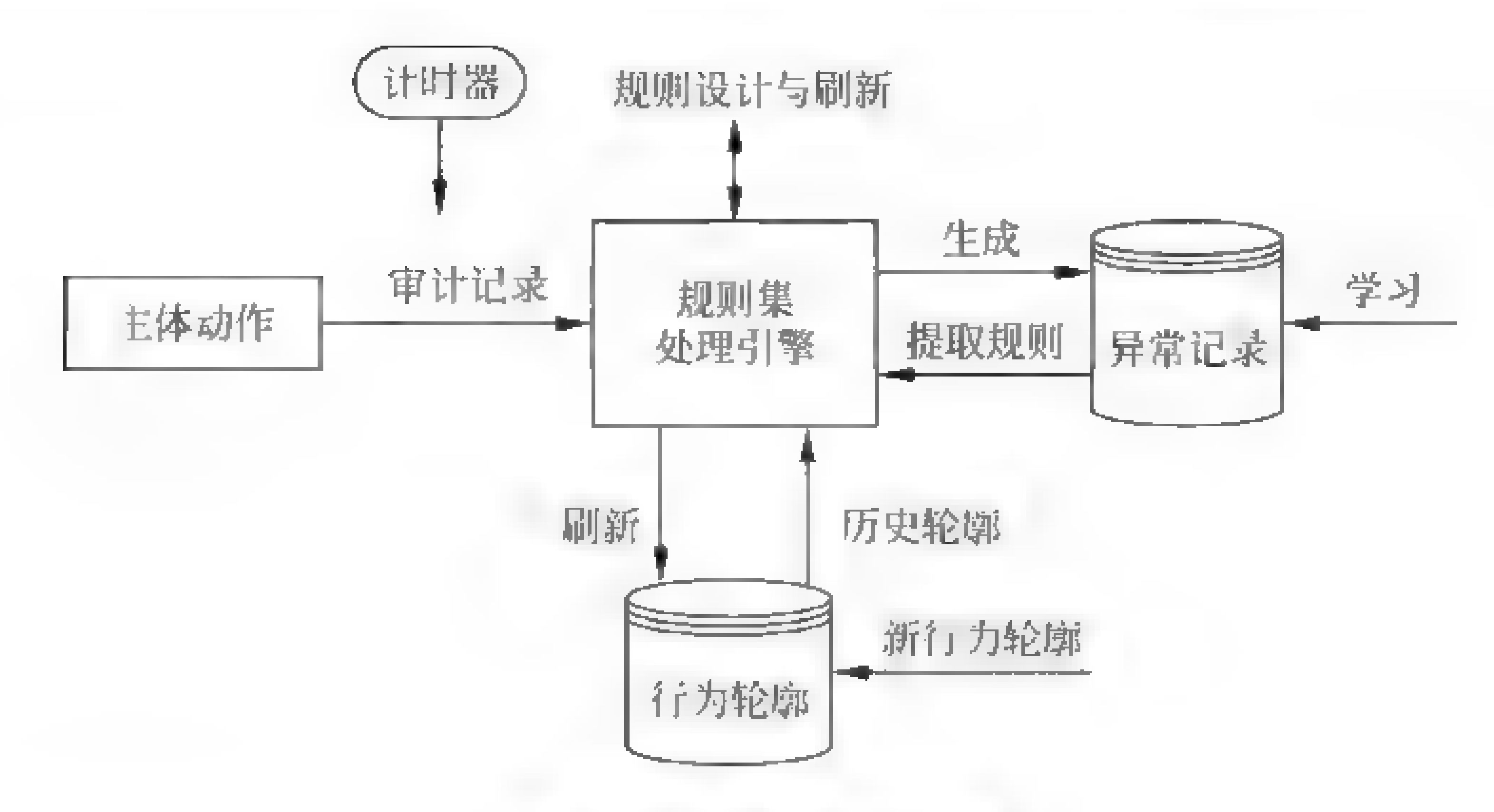


图 7-3　实时入侵检测系统

间的时间)和资源测度(即在一段时间内某个动作消耗的资源量,如程序所占用的 CPU 时间),并为这些度量(看成是随机变量)引进了以下 5 种可能的统计模型。

(1) 操作模型。在检测时把(随机变量的)一个观察值和预先确定的门限值相比较,以确定是否异常。例如,在短时间内口令错误的次数。

(2) 均值和标准差模型。对于上述介绍的随机变量,如果在检测时发现它们的观察值落在由均值和标准差决定的置信区间之外,就认为它们是异常的。

(3) 多变量模型。它是基于对两个或多个随机变量的相关分析,例如考虑它们的协方差矩阵。对于多个随机变量,如果实验表明,将它们结合在一起考虑,会比把它们一一分别考虑获得更强的判别能力,该模型就是恰当的。例如,一个程序所占用的 CPU 时间和 I/O 数据量。

(4) 马尔可夫过程模型。该模型只适用于事件计数器——每种不同的事件看成是一个状态变量,利用状态转换矩阵来刻画状态之间的转移概率。当一个命令序列,而不是单独一个个命令作为检测的对象时,该模型可被用来描述某些命令之间的转换。

(5) 时间序列模型。模型考虑一系列观察发生的顺序、到达时间和取值。它的优点在于能够测量行为的趋势和检测行为的逐渐但显著的转变。

不难看到,在特定场合下,模型(3)~(5)都比均值和标准差模型精确,但所付出的计算代价都大。Denning 的这些模型对之后的异常检测中正常轮廓的刻画起了重要的指导作用。

建立用户正常行为轮廓的最大挑战是鉴于用户的行为是动态的,如何相应地调整用户的行为轮廓呢? Ko 等人为特权程序的预期行为建模,提供了一种调整特权用户的行为轮廓新的解决问题的思路。他们的工作基于如下的假设:对于特权程序来说,由于它们所具有的特权,它们可被攻击者利用而导致系统的安全危害,但这些程序的预期行为应是有限和良性的;事先指定特权程序的预期行为,一旦在程序运行过程中出现与预期行为明显的偏差,则认为可能发生了攻击。刻画特权程序的预期行为的具体方法是利用一种程序描述(Program Specification)语言,该语言形式化地规定了一个进程所允许的操作。和检测用户轮廓相比,监测特权进程有几个优点:特权进程比用户进程更为危险,因为它们能访问计算机系统的更多部分;特权进程的行为有限且相对稳定。但监测特权进程也有其局限性,例

如，它很难检测到假冒者。这种“刻画特权程序的预期行为”的思想逐渐得到许多研究者的赞同，之后Forrest等人借鉴人体免疫系统的原理提出对特权进程的系统调用系列的统计分析思路，已成为当今异常检测研究的一个主要方法。

无论是刻画用户的正常轮廓还是特权进程的系统调用序列的正常轮廓，最初主要使用的方法是统计，但自20世纪90年代开始，各种机器学习方法开始陆续地应用于正常轮廓的学习和正常、异常的区分。比较有代表性的工作有神经网络、决策树、马尔可夫链和RIPPER(数据挖掘中的一种规则学习算法)等应用于基于用户正常轮廓的异常检测，隐马尔可夫模型、有限状态分析等用于基于特权进程的系统调用序列的正常轮廓的异常检测。

异常检测的优点是不需要事先具有攻击或系统安全漏洞的知识，而且有可能发现未知的渗透，它的研究还对信息的智能处理提出了许多富有挑战的课题，成为当今入侵检测研究的一大热点。然而，目前它的主要问题是误报率很高，因为偏离正常的行为和攻击之间还有相当的距离，在实践中异常检测还只能作为误用检测的补充。

2. 误用检测

误用检测基于这样的假设：合法用户的越权访问举动，可以通过事先刻画已知攻击的特征进行判定。误用检测的主要分析方法是利用专家系统技术建立专家特征库，比较当前行为和已知渗透行为特征，从而估计当前行为接近特定攻击行为的程度。通常这些规则和系统的配置有关，如主机的操作系统、所在的网络的配置等。不同于异常检测，规则不是由分析审计记录产生，而是由安全专家制定。例如，端口扫描的一种典型特征是在短时间内目标主机收到发往不同端口的TCP SYN包，如果涉及不开放的端口，那么攻击的可能性就更大了。

基于特征的误用检测模型的一个主要问题是特征选择上的局限性。首先，该技术不能检测出未知的攻击；其次，攻击者将想方设法修改攻击实现手段以绕过检测器的特征库，例如，将指令行中“空格”符号改成它的等价表示“%20”。这两个问题导致了基于特征的误用检测模型的高漏报率。对一个攻击的理想刻画应该是从一个标准形式出发，覆盖它的所有微妙变形(Subtle Variation)，而又不提高误报率，但目前还远远达不到该目标。

目前基于特征的误用检测模型主要是从单一事件中提取已知的攻击特征，如开源网络入侵检测系统Snort和目前大部分商业入侵检测产品均以此为最重要的检测方法。但这种方法产生的报警是建立在观察到由渗透者的一个攻击步骤所导致的现象的基础上的，因此又被称为“第一级”安全报警，这就导致报警的弱语义和误报的发生。为了提高对包含多个攻击步骤的复杂攻击的特征描述的准确性，降低检测的误报率，人们在多事件复杂特征检测领域进行了大量的研究工作，代表性的研究成果包括应用于SRI International的EMERALD系统的P BEST特征描述语言、美国UCSB开发的STAT系列原型系统、Purdue大学COAST实验室的IDIOT项目等。

3. 智能协议分析技术

智能协议分析技术充分利用了网络协议的高度有序性，使用这些知识可快速检测某个攻击特征的存在。与非智能化的模式相比，协议减少了误报的可能性；与模式匹配系统中传统的穷举分析方法相比，在处理数据包和会话时更迅速、有效。图7-4表达了智能协议分析技术模型。

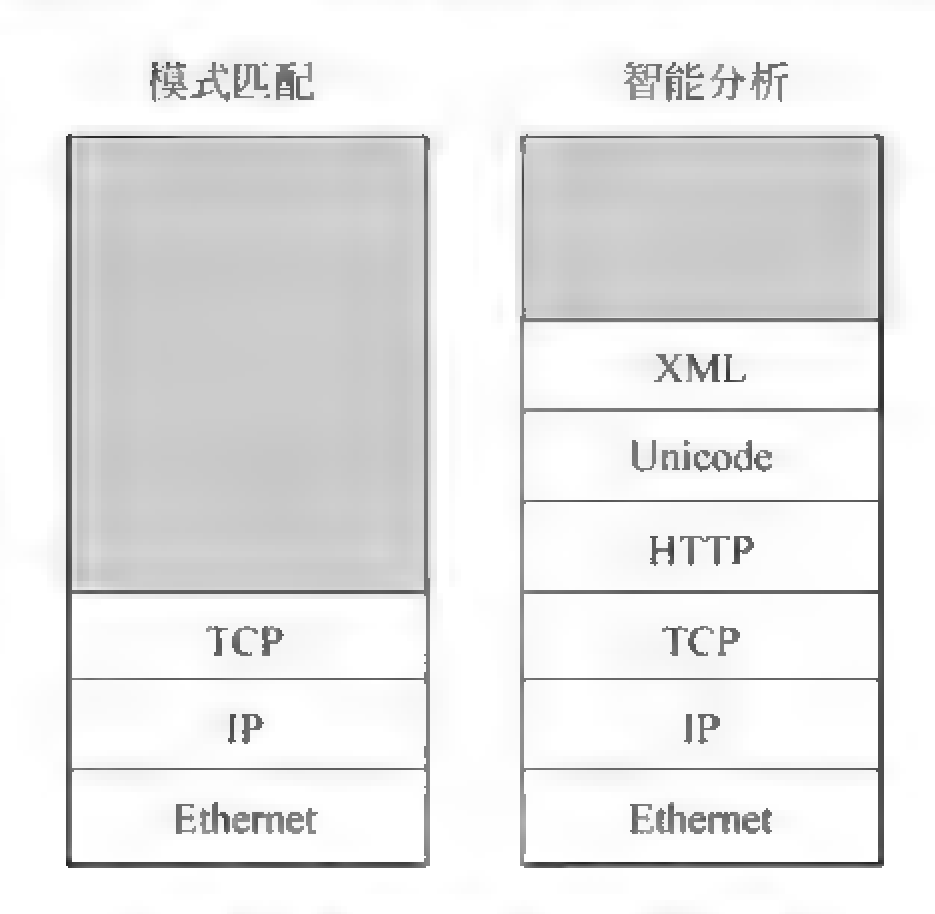


图7-4 智能协议分析技术模型

4. 会话检测技术

按照客户机与服务器之间的通信内容，把数据包重组为连续的会话流，在此基础上进行检测分析。基于数据包的入侵检测技术只对每个数据包进行检查。与基于数据包的入侵检测技术相比，会话检测技术的准确率高。

5. 统计

统计是当前产品化的入侵检测系统中常用的方法，它是一种成熟的入侵检测方法，它使入侵检测系统能够学习主体的日常行为，将那些与正常活动之间存在较大统计偏差的活动标志成异常活动。常用的入侵检测统计模型为操作模型、方差、计算参数的方差、多元模型、马尔可夫过程模型和时间序列分析。统计方法最大的优点是它可以"学习"用户的使用习惯，从而具有较高的检出率和可用性。但是，它的"学习"能力也给予入侵者机会，通过逐步"训练"使入侵事件符合正常操作的统计规律，从而使入侵检测系统无法拦截入侵者。

7.1.3 入侵检测系统的体系结构

入侵检测系统的体系结构可以分为主机型、网络型和分布式三种，其中，主机型和网络型都属于集中式系统。

1. 主机型入侵检测系统

主机型入侵检测系统位于受保护的计算机中，监控该机的运行；主要的监控源包括操作系统审计记录和系统日志。许多情况下，入侵检测系统只提供一些泛泛的报警。系统管理员可以配置入侵检测系统使得它将下列类型的变化作为可报道的安全事件：与安全相关的应用有变化，如UNIX操作系统中文件系统完整性检查软件工具Tripwire；存放关键数据的文件夹发生变化等。一旦配置得当，主机型入侵检测系统能够比较可靠地工作。

基于主机的入侵检测系统具备如下特点。

(1) 精确，可以精确地判断入侵事件。

(2) 高级，可以判断应用层的入侵事件。

(3) 对入侵时间立即进行反应。

(4) 针对不同操作系统的特点。

(5) 占用主机的宝贵资源。

2. 网络型入侵检测系统

网络型入侵检测系统的任务是在网络数据中发现攻击的特征或异常行为。局域网普遍采用的是基于广播机制的以太网协议,该协议保证传输的数据包能被统一冲突域内的所有主机接收,基于网络的入侵检测正是利用了以太网的这一特性。详细地说,以太网卡通常有正常模式和杂收模式两种。在正常模式下主机仅处理以本机为目标的数据包,而在杂收模式下网卡可以接收所处网段内传输的所有数据包,不管这些数据包的目的地址是否为本机。基于网络的入侵检测系统必须利用以太网卡的杂收模式,通过抓包工具,获得经过所处网段的所有数据信息,从而实现获得网络数据的功能。

网络型入侵检测系统监控整个网段的网络数据流,因此与主机型入侵检测系统相比,需要复杂的配置和维护,同时,网络型入侵检测系统也比主机型入侵检测系统更容易产生误报,但网络型入侵检测系统擅长对付基于网络协议的攻击手段。

基于网络的入侵检测系统具备如下特点。

(1) 能够监视经过本网段的任何活动。

(2) 实时网络监视。

(3) 监视粒度更细致。

(4) 精确度较差。

(5) 防入侵欺骗的能力较差。

(6) 交换网络环境难于配置。

3. 分布式入侵检测系统

主机型和网络型入侵检测系统在检测攻击方面各有千秋:网络型入侵检测系统擅长对付基于网络协议的攻击手段,如 SYN Flood、Ping of Death 等,而如果要精确地检测出一些常见的攻击,如缓冲区溢出,则离不开主机上的审计记录,因此对一个网段的保护需要两种入侵检测系统的合作。同时,对于大型或复杂的网络,或协作的攻击,如分布式拒绝服务攻击,需要多个检测器之间的协作,这些因素导致了分布式入侵检测系统的诞生和发展。

美国加州大学戴维斯分校研制的 DIDS(Distributed Intrusion Detection System)是最早开发的分布式入侵检测系统,图 7-5 是它的整体结构,主要由以下三个部分组成。

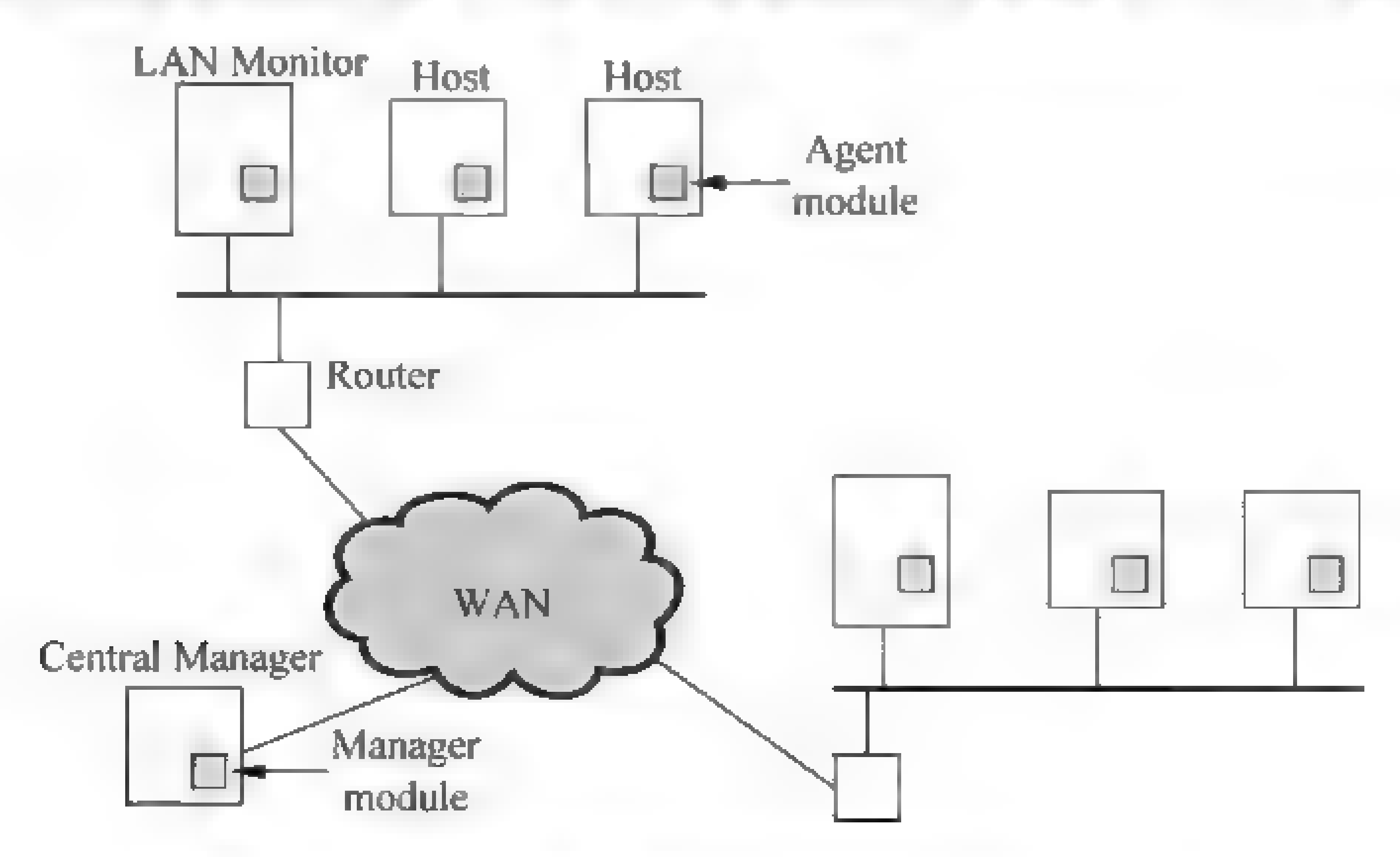


图 7-5 分布式入侵检测系统整体结构图

(1) 主机代理模块：搜集有关主机安全事件的数据，并将数据传递给中心管理员。

(2) 局域网监视代理模块：运作方式和主机相同，但它分析局域网的流量，然后将结果报告给中心管理员。

(3) 中心管理员模块：接收上述两个模块送来的报告，对它们进行综合处理，以判断是否存在入侵。

上述结构与操作系统和审计系统的具体实现无关。代理截获由原审计收集系统产生的每个审计记录，通过过滤处理，只保留与安全性有关的记录。这些记录按主机审计记录（HAR）格式重新组装，然后代理分三层分析可疑活动的记录。在最底层，代理扫描出与以前事件截然不同的事件，如失败的文件访问或改变文件访问控制等；再上一层，代理查找事件序列，如已知攻击模式；最后，代理根据用户正常轮廓查找每个用户的异常行为，如程序执行次数或文件访问次数等。当检测到可疑行动时，就向中心管理员发出警报，然后中心管理员使用专家系统进行推导。中心管理员也会要求单个主机提供 HAR 副本，与其他代理进行关联。局域网监视代理也向中心管理员提供信息。该模块审计主机之间的连接、采用的服务和网络流量，以搜索重大的事件，如网络负载突然变化，使用与安全性相关的服务等。

可以看出 DIDS 的结构非常通用和灵活，可以将检测系统从单机推广到一个可以协作的系统，从而对许多站点和网络的活动进行综合处理，对抗如分布式拒绝服务攻击等。

7.1.4　入侵检测系统的发展

入侵检测系统作为网络安全架构中的重要一环，其重要地位有目共睹。随着技术的不断完善和更新，入侵检测系统正呈现出新的发展态势。IDS(Intrusion Detection System，入侵防御系统）的出现，应该说是入侵检测系统技术的一种新发展趋势。

IDS 虽然存在一些缺陷，但换个角度可以看到，各种相关网络安全的黑客和病毒都是依托网络平台进行的。如果在网络平台上就能切断黑客和病毒的传播途径，那么就能更好地保证安全。这样，网络设备与 IDS 设备联动就应运而生了。

IDS 与网络交换设备联动，是指交换机或防火墙在运行的过程中，将各种数据流的信息上报给安全设备，IDS 可根据上报信息和数据流内容进行检测，在发现网络安全事件的时候，进行有针对性的动作，并将这些对安全事件反应的动作发送到交换机或防火墙上，由交换机或防火墙实现精确端口的关闭和断开，由此即产生了 IPS 的概念。

可认为 IPS 就是防火墙加上入侵检测系统。IPS 技术在 IDS 监测的功能上又增加了主动响应的功能，力求做到一旦发现有攻击行为，立即响应，主动切断连接。它的部署方式不像 IDS 并联在网络中，而是以串联的方式接入网络中，其功能示意图如图 7 6 所示。

也有厂商提出了 IMS(Intrusion Management System，入侵管理系统）。IMS 是一个过程，在行为未发生前要考虑网络中有什么漏洞，判断有可能会形成什么攻击行为和面临的入侵危险；在行为发生时或即将发生时，不仅要检出入侵行为，还要主动阻断，终止入侵行为；在入侵行为发生后，还要深层次分析入侵行为，通过关联分析，来判断是否还会出现下一个攻击行为。

可以看到，在入侵检测技术发展的同时，入侵技术也在更新，黑客已经将如何绕过 IDS 或攻击 IDS 作为研究重点。高速网络，尤其是交换技术的发展以及通过加密信道的数据通信，使得通过共享网段侦听的网络数据采集方法显得不足，而大量的通信量对数据分析也提

出了新的要求。随着信息系统对一个国家的社会生产与国民经济的影响越来越重要，信息战已逐步被各个国家重视，信息战中的主要攻击"武器"之一就是网络的入侵技术，信息战的防御主要包括"保护"、"检测"与"响应"，入侵检测则是其中"检测"与"响应"环节不可缺少的部分。近年来，入侵检测技术有下述主要发展方向。

(1) 分布式入侵检测与通用入侵检测架构。传统的 IDS 一般局限于单一的主机或网络架构，对异构系统及大规模的网络的监测明显不足。同时，不同的 IDS 之间不具备协同工作能力。为解决这一问题，需要分布式入侵检测技术与通用入侵检测架构。

(2) 应用层入侵检测。许多入侵的语义只有在应用层才能理解，而目前的 IDS 仅能检测 Web 之类的通用协议，不能处理如 Lotus Notes、数据库系统等其他的应用系统。许多基于客户、服务器结构、中间件技术及对象技术的大型应用，需要应用层的入侵检测保护。

(3) 智能的入侵检测。入侵方法越来越多样化与综合化，尽管智能体、神经网络与遗传算法已在入侵检测领域中应用研究，但这只是一些尝试性的研究工作，需要对智能化的 IDS 加以进一步的研究，以解决其自学习与自适应能力。

(4) 入侵检测的评测方法。用户需对众多的 IDS 进行评价，评价指标包括 IDS 检测范围、系统资源占用、IDS 自身的可靠性与稳定性。设计通用的入侵检测、评估方法与平台，实现对多种 IDS 的检测已成为当前 IDS 的另一重要研究与发展领域。

(5) 与其他网络安全技术相结合。结合防火墙、PKIX、安全电子交易 SET 等新的网络安全与电子商务技术，提供完整的网络安全保障。

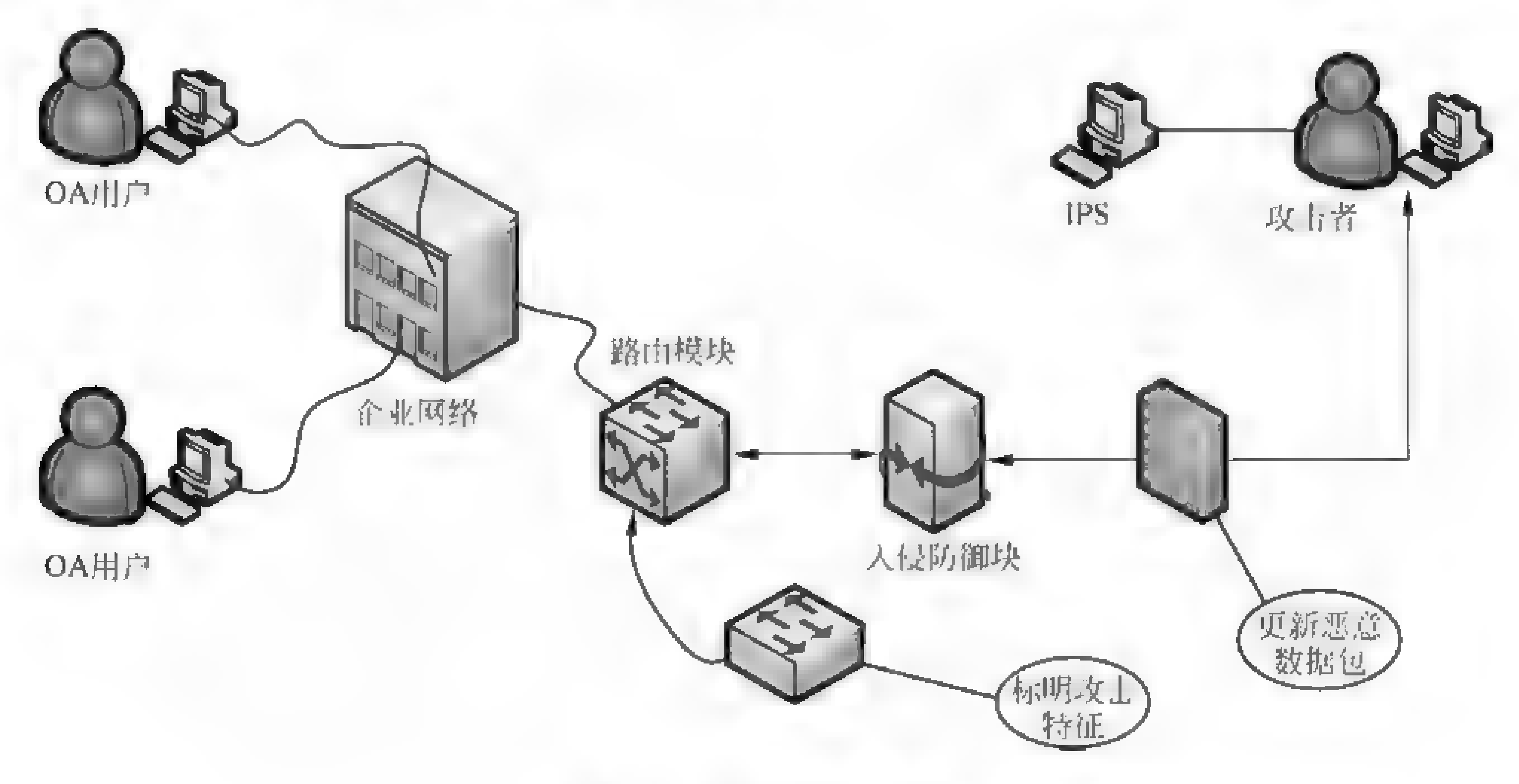


图 7-6 IPS 功能示意图

7.2 实例编程——基于 KDD 数据集及 K-Means 建立入侵检测模型

本编程训练要求以 KDD 作为基准数据集，采用经典聚类算法 K-Means 实现入侵检测。在对 KDD Cup 99 数据集的预处理之后，使用 K-Means 算法对训练数据集进行多次

聚类分析，生成聚类树，建立入侵检测模型，然后读取测试数据集的数据，预测每条记录的类别。

7.2.1 KDD CUP 99 数据集

1. KDD 概述

KDD 是数据挖掘与知识发现(Data Mining and Knowledge Discovery)的简称。KDD CUP 是由 ACM(Association for Computing Machiner)的 SIGKDD(Special Interest Group on Knowledge Discovery and Data Mining)组织的年度竞赛。

"KDD CUP 99 dataset"就是 KDD 竞赛在 1999 年举行时采用的数据集。

1998 年，美国国防部高级规划署(DARPA)在 MIT 林肯实验室进行了一项入侵检测评估项目。林肯实验室建立了模拟美国空军局域网的一个网络环境，收集了 9 周时间的 TCPdump(*)网络连接和系统审计数据，仿真各种用户类型、各种不同的网络流量和攻击手段，使它就像一个真实的网络环境。这些 TCPdump 采集的原始数据被分为两个部分：7 周时间的训练数据(**)大概包含五百多万个网络连接记录，剩下的两周时间的测试数据大概包含两百万个网络连接记录。

一个网络连接定义为在某个时间内从开始到结束的 TCP 数据包序列，并且在这段时间内，数据在预定义的协议下(如 TCP、UDP)从源 IP 地址到目的 IP 地址的传递。每个网络连接被标记为正常(Normal)或异常(Attack)，异常类型被细分为 4 大类共 39 种攻击类型，其中 22 种攻击类型出现在训练集中，另有 17 种未知攻击类型出现在测试集中。

4 种异常类型分别如下。

(1) DOS，拒绝服务攻击，例如 ping-of-death，syn flood，smurf 等。

(2) R2L，来自远程主机的未授权访问，例如 guessing password。

(3) U2R，未授权的本地超级用户特权访问，例如 buffer overflow attacks。

(4) PROBING，端口监视或扫描，例如 port-scan，ping-sweep 等。

随后，来自哥伦比亚大学的 Sal Stolfo 教授和来自北卡罗莱纳州立大学的 Wenke Lee 教授采用数据挖掘等技术对以上的数据集进行特征分析和数据预处理，形成了一个新的数据集。该数据集用于 1999 年举行的 KDD CUP 竞赛中，成为著名的 KDD99 数据集。虽然年代有些久远，但 KDD99 数据集仍然是网络入侵检测领域的事实 Benckmark，为基于计算智能的网络入侵检测研究奠定了基础。

2. 数据特征

KDD99 数据集中每个连接(*)用 41 个特征来描述：

```
2,tcp,smtp,SF,1684,363,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,1,0.00,0.00,0.00,0.00,1.00,
0.00,0.00,104,66,0.63,0.03,0.01,0.00,0.00,0.00,0.00,0.00,normal.
0,tcp,private,REJ,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,38,1,0.00,0.00,1.00,1.00,0.03,
0.55,0.00,208,1,0.00,0.11,0.18,0.00,0.01,0.00,0.42,1.00,portsweep.
0,tcp,smtp,SF,787,329,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,1,0.00,0.00,0.00,0.00,1.00,
0.00,0.00,76,117,0.49,0.08,0.01,0.02,0.00,0.00,0.00,0.00,normal.
```

上面是数据集中的三条记录，以 CSV 格式写成，加上最后的标记，一共有 42 项，其中前 41 项特征分为 4 大类，下面按顺序解释各个特征的含义。

1）TCP 连接基本特征（共 9 种）

基本连接特征包含一些连接的基本属性，如连续时间、协议类型、传送的字节数等。

duration：连接持续时间，以秒为单位，连续类型，范围是[0，58 329]。它的定义是从TCP 连接以三次握手建立算起，到 FIN ACK 连接结束为止的时间；若为 UDP 类型，则将每个 UDP 数据包作为一条连接。数据集中出现大量的 duration=0 的情况，是因为该条连接的持续时间不足 1s。

protocol_type：协议类型，离散类型，共有三种——TCP，UDP，ICMP。

service：目标主机的网络服务类型，离散类型，共有 70 种——aol，auth，bgp，courier，csnet_ns，ctf，daytime，discard，domain，domain_u，echo，eco_i，ecr_i，efs，exec，finger，ftp，ftp_data，gopher，harvest，hostnames，http，http_2784，http_443，http_8001，imap4，IRC，iso_tsap，klogin，kshell，ldap，link，login，mtp，name，netbios_dgm，netbios_ns，netbios_ssn，netstat，nnsp，nntp，ntp_u，other，pm_dump，pop_2，pop_3，printer，private，red_i，remote_job，rje，shell，smtp，sql_net，ssh，sunrpc，supdup，systat，telnet，tftp_u，tim_i，time，urh_i，urp_i，uucp，uucp_path，vmnet，whois，X11，Z39_50。

flag：连接正常或错误的状态，离散类型，共 11 种——OTH，REJ，RSTO，RSTOS0，RSTR，S0，S1，S2，S3，SF，SH。它表示该连接是否按照协议要求开始或完成。例如，SF 表示连接正常建立并终止；S0 表示只接到了 SYN 请求数据包，而没有后面的 SYN ACK。其中，SF 表示正常，其他 10 种都是 error。

src_bytes：从源主机到目标主机的数据的字节数，连续类型，范围是[0，1 379 963 888]。

dst_bytes：从目标主机到源主机的数据的字节数，连续类型，范围是[0，1 309 937 401]。

land：若连接来自/送达同一个主机/端口则为 1，否则为 0，离散类型，0 或 1。

wrong_fragment：错误分段的数量，连续类型，范围是[0，3]。

urgent：加急包的个数，连续类型，范围是[0，14]。

2）TCP 连接的内容特征（共 13 种）

对于 U2R 和 R2L 之类的攻击，由于它们不像 DoS 攻击那样在数据记录中具有频繁序列模式，而一般都是嵌入在数据包的数据负载里面，单一的数据包和正常连接没有什么区别。为了检测这类攻击，Wenke Lee 等从数据内容里面抽取了部分可能反映入侵行为的内容特征，如登录失败的次数等。

hot：访问系统敏感文件和目录的次数，连续类型，范围是[0，101]。例如访问系统目录，建立或执行程序等。

num_failed_logins：登录尝试失败的次数，连续类型，范围是[0，5]。

logged_in：成功登录则为 1，否则为 0，离散类型，0 或 1。

num_compromised：compromised 条件（**）出现的次数，连续类型，范围是[0，7479]。

root_shell：若获得 root shell 则为 1，否则为 0，离散类型，0 或 1。root_shell 是指获得超级用户权限。

su_attempted：若出现"su root"命令则为 1，否则为 0，离散类型，0 或 1。

num_root：root 用户访问次数，连续类型，范围是[0，7468]。

num_file_creations：文件创建操作的次数，连续类型，范围是[0，100]。

num_shells：使用 shell 命令的次数，连续类型，范围是[0，5]。

num_access_files：访问控制文件的次数，连续类型，范围是[0,9]。例如，对/etc/passwd或.rhosts文件的访问。

num_outbound_cmds：一个FTP会话中出现连接的次数，连续类型，0。数据集中这一特征出现次数为0。

is_hot_login：登录是否属于“hot”列表（***），是为1，否则为0，离散类型，0或1。例如，超级用户或管理员登录。

is_guest_login：若是guest登录则为1，否则为0，离散类型，0或1。

3）基于时间的网络流量统计特征（共9种，23～31）

由于网络攻击事件在时间上有很强的关联性，因此统计出当前连接记录与之前一段时间内的连接记录之间存在的某些联系，可以更好地反映连接之间的关系。这类特征又分为两种集合：一个是“same host”特征，只观察在过去两秒内与当前连接有相同目标主机的连接，例如相同的连接数，在这些相同连接与当前连接有相同的服务的连接，等等；另一个是“same service”特征，只观察过去两秒内与当前连接有相同服务的连接，例如这样的连接有多少个，其中有多少出现SYN错误或者REJ错误。

count：过去两秒内，与当前连接具有相同的目标主机的连接数，连续类型，范围是[0,511]。

srv_count：过去两秒内，与当前连接具有相同服务的连接数，连续类型，范围是[0,511]。

serror_rate：过去两秒内，在与当前连接具有相同目标主机的连接中，出现SYN错误的连接的百分比，连续类型，范围是[0.00,1.00]。

srv_serror_rate：过去两秒内，在与当前连接具有相同服务的连接中，出现SYN错误的连接的百分比，连续类型，范围是[0.00,1.00]。

rerror_rate：过去两秒内，在与当前连接具有相同目标主机的连接中，出现REJ错误的连接的百分比，连续类型，范围是[0.00,1.00]。

srv_rerror_rate：过去两秒内，在与当前连接具有相同服务的连接中，出现REJ错误的连接的百分比，连续类型，范围是[0.00,1.00]。

same_srv_rate：过去两秒内，在与当前连接具有相同目标主机的连接中，与当前连接具有相同服务的连接的百分比，连续类型，范围是[0.00,1.00]。

diff_srv_rate：过去两秒内，在与当前连接具有相同目标主机的连接中，与当前连接具有不同服务的连接的百分比，连续类型，范围是[0.00,1.00]。

srv_diff_host_rate：过去两秒内，在与当前连接具有相同服务的连接中，与当前连接具有不同目标主机的连接的百分比，连续类型，范围是[0.00,1.00]。

注：这一大类特征中，23、25、27、29、30这5个特征是“same host”特征，前提都是与当前连接具有相同目标主机的连接；24、26、28、31这4个特征是“same service”特征，前提都是与当前连接具有相同服务的连接。

4）基于主机的网络流量统计特征（共10种，32～41）

基于时间的流量统计只是在过去两秒的范围内统计与当前连接之间的关系，而在实际入侵中，有些Probing攻击使用慢速攻击模式来扫描主机或端口，当它们扫描的频率大于两秒的时候，基于时间的统计方法就无法从数据中找到关联。所以Wenke Lee等按照目标主机进行分类，使用一个具有100个连接的时间窗，统计当前连接之前100个连接记录中与当前连接具有相同目标主机的统计信息。

dst_host_count：前 100 个连接中，与当前连接具有相同目标主机的连接数，连续类型，范围是[0,255]。

dst_host_srv_count：前 100 个连接中，与当前连接具有相同目标主机相同服务的连接数，连续类型，范围是[0,255]。

dst_host_same_srv_rate：前 100 个连接中，与当前连接具有相同目标主机相同服务的连接所占的百分比，连续类型，范围是[0.00,1.00]。

dst_host_diff_srv_rate：前 100 个连接中，与当前连接具有相同目标主机不同服务的连接所占的百分比，连续类型，范围是[0.00,1.00]。

dst_host_same_src_port_rate：前 100 个连接中，与当前连接具有相同目标主机相同源端口的连接所占的百分比，连续类型，范围是[0.00,1.00]。

dst_host_srv_diff_host_rate：前 100 个连接中，与当前连接具有相同目标主机相同服务的连接中，与当前连接具有不同源主机的连接所占的百分比，连续类型，范围是[0.00,1.00]。

dst_host_serror_rate：前 100 个连接中，与当前连接具有相同目标主机的连接中，出现 SYN 错误的连接所占的百分比，连续类型，范围是[0.00,1.00]。

dst_host_srv_serror_rate：前 100 个连接中，与当前连接具有相同目标主机相同服务的连接中，出现 SYN 错误的连接所占的百分比，连续类型，范围是[0.00,1.00]。

dst_host_rerror_rate：前 100 个连接中，与当前连接具有相同目标主机的连接中，出现 REJ 错误的连接所占的百分比，连续类型，范围是[0.00,1.00]。

dst_host_srv_rerror_rate：前 100 个连接中，与当前连接具有相同目标主机相同服务的连接中，出现 REJ 错误的连接所占的百分比，连续类型，范围是[0.00,1.00]。

7.2.2 K-Means 算法原理

1. 聚类分析

聚类分析又称群分析，它是研究(样品或指标)分类问题的一种统计分析方法，同时也是数据挖掘的一个重要算法。

聚类(Cluster)分析是由若干模式(Pattern)组成的，通常，模式是一个度量(Measurement)的向量，或者是多维空间中的一个点。

聚类分析以相似性为基础，在一个聚类中的模式之间比不在同一聚类中的模式之间具有更多的相似性。

2. K-Means 算法

K-Means 算法是很典型的基于距离的聚类算法，采用距离作为相似性的评价指标，即认为两个对象的距离越近，其相似度就越大。该算法认为簇是由距离靠近的对象组成的，因此把得到紧凑且独立的簇作为最终目标。

k 个初始类聚类中心点的选取对聚类结果具有较大的影响，因为在该算法第一步中是随机地选取任意 k 个对象作为初始聚类的中心，初始地代表一个簇。该算法在每次迭代中对数据集中剩余的每个对象，根据其与各个簇中心的距离将每个对象重新赋给最近的簇。当考察完所有数据对象后，一次迭代运算完成，新的聚类中心被计算出来。如果在一次迭代前后，J 的值没有发生变化，说明算法已经收敛。

算法过程如下。

(1) 从 n 个文档随机选取 k 个文档作为质心；

(2) 对剩余的每个文档测量其到每个质心的距离，并把它归到最近的质心的类；

(3) 重新计算已经得到的各个类的质心；

(4) 迭代(2)、(3)步直至新的质心与原质心相等或小于指定阈值，算法结束。

具体如下。

输入：k，data[n]；

(1) 选择 k 个初始中心点，例如 c[0]=data[0]，…，c[$k-1$]=data[$k-1$]；

(2) 对于 data[0]…data[n]，分别与 c[0]…c[$k-1$]比较，假定与 c[i]差值最少，就标记为 i；

(3) 对于所有标记为 i 点，重新计算 c[i]={所有标记为 i 的 data[j]之和}/标记为 i 的个数；

(4) 重复(2)、(3)，直到所有 c[i]值的变化小于给定阈值。

3. 工作原理及处理流程

K-Means 算法接受输入量 k；然后将 n 个数据对象划分为 k 个聚类以便使得所获得的聚类满足：同一聚类中的对象相似度较高；而不同聚类中的对象相似度较小。聚类相似度是利用各聚类中对象的均值所获得一个"中心对象"(引力中心)来进行计算的。具体算法如下。

输入：聚类个数 k，以及包含 n 个数据对象的数据库。

输出：满足方差最小标准的 k 个聚类。

处理流程如下。

(1) 从 n 个数据对象中任意选择 k 个对象作为初始聚类中心；

(2) 根据每个聚类对象的均值(中心对象)，计算每个对象与这些中心对象的距离；并根据最小距离重新对相应对象进行划分；

(3) 重新计算每个(有变化)聚类的均值(中心对象)；

(4) 循环(2)、(3)直到每个聚类不再发生变化为止。

K-Means 算法示意图如图 7-7 所示。

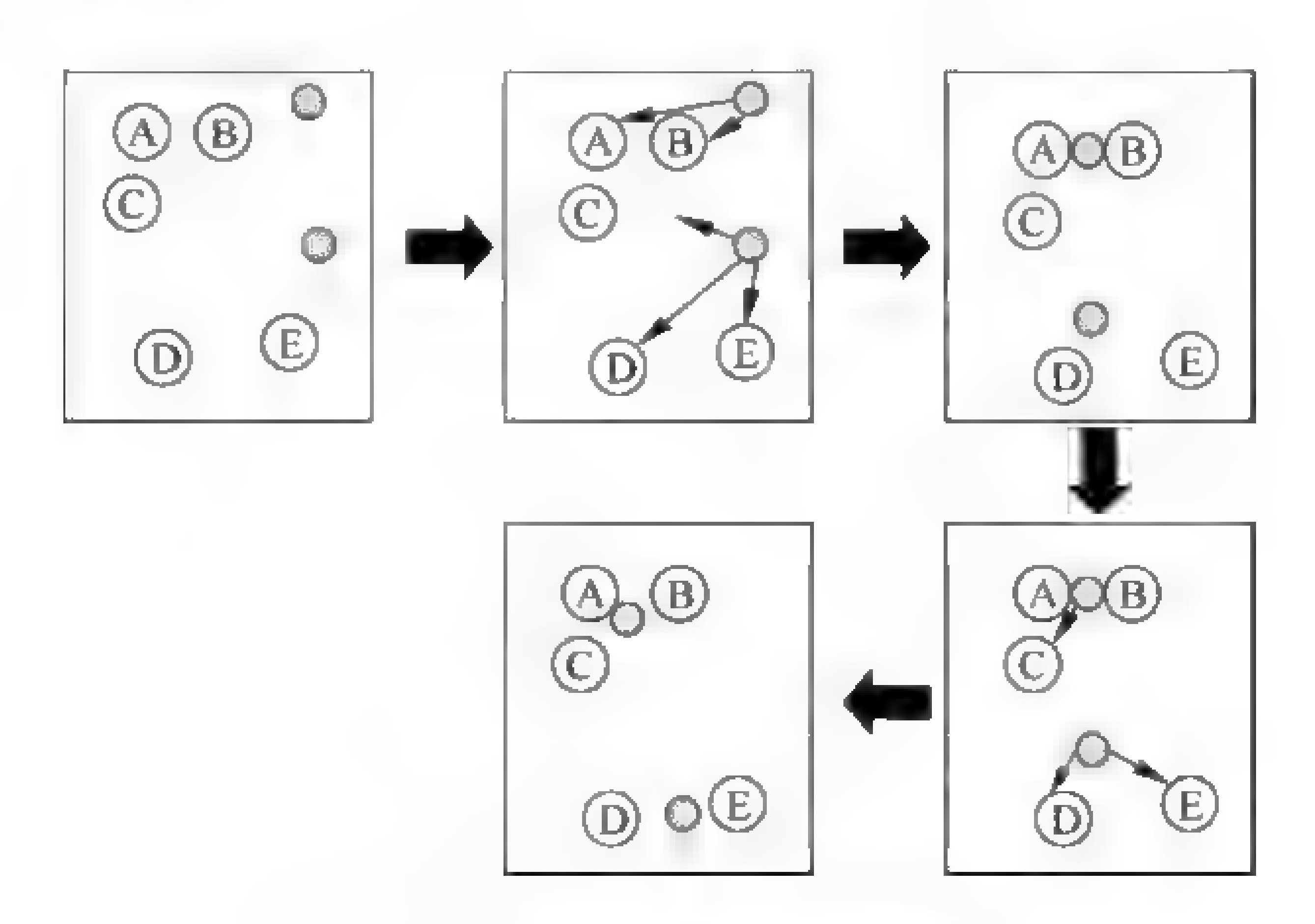


图 7-7 K-Means 算法示意图

7.2.3 K-Means 算法代码实现

1. k-means.h

```
#pragma once
#include <fstream>

class KMeans
{
public:
enum InitMode
{
    InitRandom,
    InitManual,
    InitUniform,
};

KMeans(int dimNum = 1, int clusterNum = 1);
~KMeans();

void SetMean(int i, const double * u){ memcpy(m_means[i], u, sizeof(double) * m_dimNum); }
void SetInitMode(int i)          { m_initMode = i; }
void SetMaxIterNum(int i)        { m_maxIterNum = i; }
void SetEndError(double f)       { m_endError = f; }

double * GetMean(int i)   { return m_means[i]; }
int GetInitMode()         { return m_initMode; }
int GetMaxIterNum()           { return m_maxIterNum; }
double GetEndError()      { return m_endError; }


/*  SampleFile: <size><dim><data>...
    LabelFile:  <size><label>...
*/
void Cluster(const char * sampleFileName, const char * labelFileName);
void Init(std::ifstream& sampleFile);
void Init(double *data, int N);
void Cluster(double *data, int N, int *Label);
friend std::ostream& operator<<(std::ostream& out, KMeans& kmeans);

private:
int m_dimNum;
int m_clusterNum;
double ** m_means;

int m_initMode;
int m_maxIterNum;            //The stopping criterion regarding the number of iterations
double m_endError;           //The stopping criterion regarding the error
```

```
double GetLabel(const double* x, int* label);
double CalcDistance(const double* x, const double* u, int dimNum);
};
```

2. k-means. cpp

```
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <iostream>
#include "k-means.h"
using namespace std;


KMeans::KMeans(int dimNum, int clusterNum)
{
m_dimNum = dimNum;
m_clusterNum = clusterNum;

m_means = new double*[m_clusterNum];
for(int i = 0; i < m_clusterNum; i++)
{
    m_means[i] = new double[m_dimNum];
    memset(m_means[i], 0, sizeof(double) * m_dimNum);
}

m_initMode = InitRandom;
m_maxIterNum = 100;
m_endError = 0.001;
}

KMeans::~KMeans()
{
for(int i = 0; i < m_clusterNum; i++)
{
    delete[] m_means[i];
}
delete[] m_means;
}

void KMeans::Cluster(const char* sampleFileName, const char* labelFileName)
{
//Check the sample file
ifstream sampleFile(sampleFileName, ios_base::binary);
assert(sampleFile);

int size = 0;
int dim = 0;
sampleFile.read((char*)&size, sizeof(int));
sampleFile.read((char*)&dim, sizeof(int));
assert(size >= m_clusterNum);
```

```
assert(dim == m_dimNum);

//Initialize model
Init(sampleFile);

//Recursion
double* x = new double[m_dimNum];       //Sample data
int label = -1;                          //Class index
double iterNum = 0;
double lastCost = 0;
double currCost = 0;
int unchanged = 0;
bool loop = true;
int* counts = new int[m_clusterNum];
double** next_means = new double*[m_clusterNum];    //New model for reestimation
for(int i = 0; i < m_clusterNum; i++)
{
    next_means[i] = new double[m_dimNum];
}

while(loop)
{
    //clean buffer for classification
    memset(counts, 0, sizeof(int) * m_clusterNum);
    for(int i = 0; i < m_clusterNum; i++)
    {
        memset(next_means[i], 0, sizeof(double) * m_dimNum);
    }

    lastCost = currCost;
    currCost = 0;

    sampleFile.clear();
    sampleFile.seekg(sizeof(int) * 2, ios_base::beg);

    //Classification
    for(int i = 0; i < size; i++)
    {
        sampleFile.read((char*)x, sizeof(double) * m_dimNum);
        currCost += GetLabel(x, &label);

        counts[label]++;
        for(int d = 0; d < m_dimNum; d++)
        {
            next_means[label][d] += x[d];
        }
    }
    currCost /= size;

    //Reestimation
```

```
        for(int i = 0; i < m_clusterNum; i++)
        {
            if(counts[i] > 0)
            {
                for(int d = 0; d < m_dimNum; d++)
                {
                    next_means[i][d] /= counts[i];
                }
                memcpy(m_means[i], next_means[i], sizeof(double) * m_dimNum);
            }
        }

        //Terminal conditions
        iterNum++;
        if(fabs(lastCost - currCost) < m_endError * lastCost)
        {
            unchanged++;
        }
        if(iterNum >= m_maxIterNum || unchanged >= 3)
        {
            loop = false;
        }
        //DEBUG
        //cout << "Iter: " << iterNum << ", Average Cost: " << currCost << endl;
    }

    //Output the label file
    ofstream labelFile(labelFileName, ios_base::binary);
    assert(labelFile);

    labelFile.write((char * )&size, sizeof(int));
    sampleFile.clear();
    sampleFile.seekg(sizeof(int) * 2, ios_base::beg);

    for(int i = 0; i < size; i++)
    {
        sampleFile.read((char * )x, sizeof(double) * m_dimNum);
        GetLabel(x, &label);
        labelFile.write((char * )&label, sizeof(int));
    }

    sampleFile.close();
    labelFile.close();

    delete[] counts;
    delete[] x;
    for(int i = 0; i < m_clusterNum; i++)
    {
        delete[] next_means[i];
    }
```

```
delete[] next_means;
}

//N 为特征向量数
void KMeans::Cluster(double * data, int N, int * Label)
{
int size = 0;
size = N;
assert(size >= m_clusterNum);

//Initialize model
Init(data,N);

//Recursion
double* x = new double[m_dimNum];                    //Sample data
int label = -1;                                      //Class index
double iterNum = 0;
double lastCost = 0;
double currCost = 0;
int unchanged = 0;
bool loop = true;
int* counts = new int[m_clusterNum];
double** next_means = new double*[m_clusterNum]; //New model for reestimation
for(int i = 0; i < m_clusterNum; i++)
{
    next_means[i] = new double[m_dimNum];
}

while(loop)
{
    //clean buffer for classification
    memset(counts, 0, sizeof(int) * m_clusterNum);
    for(int i = 0; i < m_clusterNum; i++)
    {
        memset(next_means[i], 0, sizeof(double) * m_dimNum);
    }
    lastCost = currCost;
    currCost = 0;
    //Classification
    for(int i = 0; i < size; i++)
    {
        for(int j = 0; j < m_dimNum; j++)
            x[j] = data[i * m_dimNum + j];
        currCost += GetLabel(x, &label);
        counts[label]++;
        for(int d = 0; d < m_dimNum; d++)
        {
            next_means[label][d] += x[d];
        }
    }
```

```
        currCost /= size;

        //Reestimation
        for(int i = 0; i < m_clusterNum; i++)
        {
            if(counts[i] > 0)
            {
                for(int d = 0; d < m_dimNum; d++)
                {
                    next_means[i][d] /= counts[i];
                }
                memcpy(m_means[i], next_means[i], sizeof(double) * m_dimNum);
            }
        }

        //Terminal conditions
        iterNum++;
        if(fabs(lastCost - currCost) < m_endError * lastCost)
        {
            unchanged++;
        }
        if(iterNum >= m_maxIterNum || unchanged >= 3)
        {
            loop = false;
        }

        //DEBUG
        //cout << "Iter: " << iterNum << ", Average Cost: " << currCost << endl;
    }

    //Output the label file
    for(int i = 0; i < size; i++)
    {
        for(int j = 0; j < m_dimNum; j++)
            x[j] = data[i * m_dimNum + j];
        GetLabel(x,&label);
        Label[i] = label;
    }
    delete[] counts;
    delete[] x;
    for(int i = 0; i < m_clusterNum; i++)
    {
        delete[] next_means[i];
    }
    delete[] next_means;
    }

    void KMeans::Init(double *data, int N)
    {
    int size = N;
```

```
if(m_initMode ==  InitRandom)
{
    int inteval = size / m_clusterNum;
    double* sample = new double[m_dimNum];
    //Seed the random - number generator with current time
    srand((unsigned)time(NULL));
    for(int i = 0; i < m_clusterNum; i++)
    {
        int select = inteval * i + (inteval - 1) * rand() / RAND_MAX;
        for(int j = 0; j < m_dimNum; j++)
            sample[j] = data[select * m_dimNum + j];
        memcpy(m_means[i], sample, sizeof(double) * m_dimNum);
    }
    delete[] sample;
}
else if(m_initMode == InitUniform)
{
    double* sample = new double[m_dimNum];

    for(int i = 0; i < m_clusterNum; i++)
    {
        int select = i * size / m_clusterNum;
        for(int j = 0; j < m_dimNum; j++)
            sample[j] = data[select * m_dimNum + j];
        memcpy(m_means[i], sample, sizeof(double) * m_dimNum);
    }
    delete[] sample;
}
else if(m_initMode == InitManual)
{
    //Do nothing
}
}

void KMeans::Init(ifstream& sampleFile)
{
int size = 0;
sampleFile.seekg(0, ios_base::beg);
sampleFile.read((char*)&size, sizeof(int));

if(m_initMode ==  InitRandom)
{
    int inteval = size / m_clusterNum;
    double* sample = new double[m_dimNum];
    //Seed the random - number generator with current time
    srand((unsigned)time(NULL));
    for(int i = 0; i < m_clusterNum; i++)
    {
        int select = inteval * i + (inteval - 1) * rand() / RAND_MAX;
```

```
        int offset = sizeof(int) * 2 + select * sizeof(double) * m_dimNum;

        sampleFile.seekg(offset, ios_base::beg);
        sampleFile.read((char *)sample, sizeof(double) * m_dimNum);
        memcpy(m_means[i], sample, sizeof(double) * m_dimNum);
    }
    delete[] sample;
}
else if(m_initMode == InitUniform)
{
    double * sample = new double[m_dimNum];

    for (int i = 0; i < m_clusterNum; i++)
    {
        int select = i * size / m_clusterNum;
        int offset = sizeof(int) * 2 + select * sizeof(double) * m_dimNum;
        sampleFile.seekg(offset, ios_base::beg);
        sampleFile.read((char *)sample, sizeof(double) * m_dimNum);
        memcpy(m_means[i], sample, sizeof(double) * m_dimNum);
    }
    delete[] sample;
}
else if(m_initMode == InitManual)
{
    //Do nothing
}
}

double KMeans::GetLabel(const double * sample, int * label)
{
double dist = -1;
for(int i = 0; i < m_clusterNum; i++)
{
    double temp = CalcDistance(sample, m_means[i], m_dimNum);
    if(temp < dist || dist == -1)
    {
        dist = temp;
        *label = i;
    }
}
return dist;
}

double KMeans::CalcDistance(const double * x, const double * u, int dimNum)
{
double temp = 0;
for(int d = 0; d < dimNum; d++)
{
    temp += (x[d] - u[d]) * (x[d] - u[d]);
}
```

```
return sqrt(temp);
}

ostream& operator <<(ostream& out, KMeans& kmeans)
{
out << "< KMeans >" << endl;
out << "< DimNum > " << kmeans.m_dimNum << " </DimNum >" << endl;
out << "< ClusterNum > " << kmeans.m_clusterNum << " </CluterNum >" << endl;

out << "< Mean >" << endl;
for(int i = 0; i < kmeans.m_clusterNum; i++)
{
    for(int d = 0; d < kmeans.m_dimNum; d++)
    {
        out << kmeans.m_means[i][d] << " ";
    }
    out << endl;
}
out << "</Mean >" << endl;
out << "</KMeans >" << endl;
return out;
}
```

3. main. cpp

```
#include < iostream >
#include "k - means.h"
using namespace std;

int main()
{
    double data[] = {
        0.0, 0.2, 0.4,
        0.3, 0.2, 0.4,
        0.4, 0.2, 0.4,
        0.5, 0.2, 0.4,
        5.0, 5.2, 8.4,
        6.0, 5.2, 7.4,
        4.0, 5.2, 4.4,
        10.3, 10.4, 10.5,
        10.1, 10.6, 10.7,
        11.3, 10.2, 10.9
    };

    const int size = 10;            //Number of samples
    const int dim = 3;              //Dimension of feature
    const int cluster_num = 4;      //Cluster number

    KMeans * kmeans = new KMeans(dim,cluster_num);
    int * labels = new int[size];
    kmeans -> SetInitMode(KMeans::InitUniform);
```

```
kmeans->Cluster(data,size,labels);

for(int i = 0; i < size; ++i)
{
    printf("%f, %f, %f belongs to %d cluster\n", data[i*dim+0], data[i*dim+1], data
[i*dim+2], labels[i]);
}

delete []labels;
delete kmeans;

    return 0;
}
```

小　结

本章对入侵检测的基本原理、主要分析模型和方法、体系结构，以及入侵检测系统的发展进行了介绍，并对 KDD CUP 99 数据集和 K Means 算法相关知识进行了描述，提供了 K Means 算法的具体代码实现。

思 考 题

1. 简要描述入侵检测的基本原理。
2. 目前入侵检测的分析模型主要有哪些?
3. 简述入侵检测系统的体系结构。
4. 查阅资料，深入理解并掌握 K-Means 算法。
5. 学习一种数据挖掘工具(例如 WEKA)，基于该工具建立入侵检测模型。

第8章　应用系统安全编程

由于应用系统的复杂性，有关应用平台的安全问题是整个安全体系中最复杂的部分。本章挑选了最常见的网络应用安全编程：安全Web服务器和安全电子邮件，基于OpenSSL开发包进行实现。

8.1　基于OpenSSL的安全Web服务器程序

8.1.1　基础知识

1. Web Server

Web Server是企业对外宣传、开展业务的重要基地。由于其重要性，成为Hacker攻击的首选目标之一。

Web Server经常成为Internet用户访问公司内部资源的通道之一，如Web Server通过中间件访问主机系统，通过数据库连接部件访问数据库，利用CGI访问本地文件系统或网络系统中的其他资源。但Web服务器越来越复杂，其被发现的安全漏洞越来越多。为了防止Web服务器成为攻击的牺牲品或成为进入内部网络的跳板，需要给予它更多的关心。

(1) Web服务器应置于防火墙保护之下。

(2) 在Web服务器上安装实时安全监控软件。

(3) 在通往Web服务器的网络路径上安装基于网络的实时入侵监控系统。

(4) 经常审查Web服务器配置情况及运行日志。

(5) 运行新的应用前，先进行安全测试，如新的CGI应用。

(6) 认证过程采用加密通信或使用X.509证书模式。

(7) 小心设置Web服务器的访问控制表。

2. SSL协议

1) SSL协议的功能

SSL(Secure Socket Layer)为Netscape所研发，用以保障在Internet上数据传输的安全，利用数据加密(Encryption)技术，可确保数据在网络上的传输过程中不会被截取及窃听。一般通用规格为40b的安全标准，美国则已推出128b的更高安全标准，但限制出境。只要3.0版本以上I.E.或Netscape浏览器即可支持SSL。当前SSL版本为3.0。它已被广泛地用于Web浏览器与服务器之间的身份认证和加密数据传输。

SSL(Secure Socket Layer，安全套接层)及其继任者传输层安全(Transport Layer Security，TLS)是为网络通信提供安全及数据完整性的一种安全协议。TLS与SSL在传输层对网络连接进行加密。SSL协议位于TCP/IP与各种应用层协议之间，为数据通信提供安全支持。SSL协议可分为两层：SSL记录协议(SSL Record Protocol)，它建立在可靠的

传输协议(如 TCP)之上,为高层协议提供数据封装、压缩、加密等基本功能的支持;SSL 握手协议(SSL Handshake Protocol),它建立在 SSL 记录协议之上,用于在实际的数据传输开始前,通信双方进行身份认证、协商加密算法、交换加密密钥等。

2) SSL 工作流程

服务器认证阶段的工作流程如下。

(1) 客户端向服务器发送一个开始信息"Hello"以便开始一个新的会话连接;

(2) 服务器根据客户的信息确定是否需要生成新的主密钥,如需要则服务器在响应客户的"Hello"信息时将包含生成主密钥所需的信息;

(3) 客户根据收到的服务器响应信息,产生一个主密钥,并用服务器的公开密钥加密后传给服务器;

(4) 服务器回复该主密钥,并返回给客户一个用主密钥认证的信息,以此让客户认证服务器。

用户认证阶段:在此之前,服务器已经通过了客户认证,这一阶段主要完成对客户的认证。经认证的服务器发送一个提问给客户,客户则返回(数字)签名后的提问和其公开密钥,从而向服务器提供认证。

SSL 协议提供的安全通道具有以下三个特性。

(1) 机密性:SSL 协议使用密钥加密通信数据。

(2) 可靠性:服务器和客户都会被认证,客户的认证是可选的。

(3) 完整性:SSL 协议会对传送的数据进行完整性检查。

从 SSL 协议所提供的服务及其工作流程可以看出,SSL 协议运行的基础是商家对消费者信息保密的承诺,这就有利于商家而不利于消费者。在电子商务初级阶段,由于运作电子商务的企业大多是信誉较高的大公司,因此这个问题还没有充分暴露出来。但随着电子商务的发展,各中小型公司也参与进来,这样在电子支付过程中的单一认证问题就越来越突出。虽然在 SSL 3.0 中通过数字签名和数字证书可实现浏览器和 Web 服务器双方的身份验证,但是 SSL 协议仍存在一些问题,比如,只能提供交易中客户与服务器间的双方认证,在涉及多方的电子交易中,SSL 协议并不能协调各方间的安全传输和信任关系。在这种情况下,Visa 和 MasterCard 两大信用卡公司组织制定了 SET 协议,为网上信用卡支付提供了全球性的标准。

3) SSL 的体系结构

SSL 的体系结构中包含两个协议子层,其中底层是 SSL 记录协议层(SSL Record Protocol Layer),高层是 SSL 握手协议层(SSL HandShake Protocol Layer)。SSL 的协议栈如图 8-1 所示,其中阴影部分即 SSL 协议。



图 8-1　SSL 协议栈

SSL记录协议层的作用是为高层协议提供基本的安全服务。SSL记录协议针对HTTP进行了特别的设计，使得超文本的传输协议HTTP能够在SSL上运行。记录封装各种高层协议，具体实施压缩解压缩、加密解密、计算和校验MAC等与安全有关的操作。

SSL握手协议层包括SSL握手协议(SSL HandShake Protocol)、SSL密码参数修改协议(SSL Change Cipher Spec Protocol)、应用数据协议(Application Data Protocol)和SSL告警协议(SSL Alert Protocol)。握手层的这些协议用于SSL管理信息的交换，允许应用协议传送数据之间相互验证，协商加密算法和生成密钥等。SSL握手协议的作用是协调客户和服务器的状态，使双方能够达到状态的同步。SSL记录协议(Record Protocol)为SSL提供以下两种服务。

(1) 保密性：利用握手协议所定义的共享密钥对SSL净荷(Payload)加密。

(2) 完整性：利用握手协议所定义的共享的MAC密钥来生成报文的鉴别码(MAC)。

4) SSL的工作过程

发送方的工作过程如下。

(1) 从上层接收要发送的数据(包括各种消息和数据)；

(2) 对信息进行分段，分成若干记录；

(3) 使用指定的压缩算法进行数据压缩(可选)；

(4) 使用指定的MAC算法生成MAC；

(5) 使用指定的加密算法进行数据加密；

(6) 添加SSL记录协议的头，发送数据。

接收方的工作过程如下。

(1) 接收数据，从SSL记录协议的头中获取相关信息；

(2) 使用指定的解密算法解密数据；

(3) 使用指定的MAC算法校验MAC；

(4) 使用压缩算法对数据解压缩(在需要时进行)；

(5) 将记录进行数据重组；

(6) 将数据发送给高层。

SSL记录协议处理的最后一个步骤是附加一个SSL记录协议的头，以便构成一个SSL记录。SSL记录协议头中包含SSL记录协议的若干控制信息。

5) SSL的会话状态

会话(Session)和连接(Connection)是SSL中两个重要的概念，在规范中定义如下。

SSL连接：用于提供某种类型的服务数据的传输，是一种点对点的关系。一般来说，连接的维持时间比较短暂，并且每个连接一定与某一个会话相关联。

SSL会话：指客户和服务器之间的一个关联关系。会话通过握手协议来创建。它定义了一组安全参数。

一次会话过程通常会发起多个SSL连接来完成任务，例如，一次网站的访问可能需要多个HTTP SSL TCP连接来下载其中的多个页面，这些连接共享会话定义的安全参数。这种共享方式可以避免为每个SSL连接单独进行安全参数的协商，而只需在会话建立时进行一次协商，提高了效率。

每一个会话(或连接)都存在一组与之相对应的状态，会话(或连接)的状态表现为一组

与其相关的参数集合，最主要的内容是与会话(或连接)相关的安全参数的集合，用会话(或连接)中的加密解密、认证等安全功能的实现。在SSL通信过程中，通信算法的状态通过SSL握手协议实现同步。

根据SSL协议的约定，会话状态由以下参数来定义。

会话标识符：由服务器选择的任意字节序列，用于标识活动的会话或可恢复的会话状态。

对方的证书：会话对方的X.509v3证书。该参数可为空。

压缩算法：在加密之前用来压缩数据的算法。

加密规约(Cipher Spec)：用于说明对大块数据进行加密采用的算法，以及计算MAC所采用的散列算法。

主密值：一个48字节长的秘密值，由客户和服务器共享。

6) 可重新开始的标识：用于指示会话是否可以用于初始化新的连接

连接状态可以用以下参数来定义。

服务器和客户机的随机数：服务器和客户机为每个连接选择的用于标识连接的字节序列。

服务器写MAC密值：服务器发送数据时，生成MAC。

使用的密钥：长度为128b。

客户机写MAC密值：服务器发送数据时，用于数据加密的密钥，长度为128b。

客户机写密钥：客户机发送数据时，用于数据加密的密钥，长度为128b。

初始化向量：当使用CBC模式的分组密文算法时，需要为每个密钥维护初始化向量。

序列号：通信的每一端都为每个连接中的发送和接收报文维持着一个序列号。

7) HTTPS介绍

HTTPS(Hypertext Transfer Protocol Secure，安全超文本传输协议)是由Netscape开发并内置于其浏览器中，用于对数据进行压缩和解压操作，并返回网络上传送回的结果。HTTPS实际上应用了Netscape的完全套接字层(SSL)作为HTTP应用层的子层。(HTTPS使用端口443，而不是像HTTP那样使用端口80来和TCP/IP进行通信)。SSL使用40位关键字作为RC4流加密算法，这对于商业信息的加密是合适的。HTTPS和SSL支持使用X.509数字认证，如果需要的话用户可以确认发送者是谁。

HTTPS是以安全为目标的HTTP通道，简单地讲就是HTTP的安全版。即HTTP下加入SSL层，HTTPS的安全基础是SSL，因此加密的详细内容请看SSL。

它是一个URI Scheme(抽象标识符体系)，句法类同http:体系，用于安全的HTTP数据传输。https:URL表明它使用了HTTP，但HTTPS存在不同于HTTP的默认端口及一个加密/身份验证层(在HTTP与TCP之间)。这个系统的最初研发由网景公司进行，提供了身份验证与加密通信方法，它被广泛用于万维网上安全敏感的通信，例如交易支付方面。

HTTPS的安全保护依赖浏览器的正确实现以及服务器软件、实际加密算法的支持。

一种常见的误解是“银行用户在线使用https:就能充分彻底地保障他们的银行卡号不被偷窃。”实际上，与服务器的加密连接中能保护银行卡号的部分，只有用户到服务器之间的连接及服务器自身，并不能绝对确保服务器自己是安全的，这点甚至已被攻击者利用，常见

例子是模仿银行域名的钓鱼攻击。少数罕见攻击在网站传输客户数据时发生，攻击者尝试窃听数据于传输中。

商业网站被人们期望迅速尽早引入新的特殊处理程序到金融网关，仅保留传输码(Transaction Number)。不过他们常常存储银行卡号在同一个数据库里。那些数据库和服务器少数情况有可能被未授权用户攻击和损害。

8.1.2 基于 OpenSSL 的安全 Web 编程实现

1. BIO 机制

BIO 机制是 OpenSSL 提供的一种高层 I/O 接口，该接口封装了几乎所有类型的 I/O 接口，如内存访问、文件访问以及 Socket 等。这使得代码的重用性大幅度提高，OpenSSL 提供 API 的复杂性也降低了很多。

基本的 BIO 函数如下。

BIO_new：新创建一个 BIO 对象。

BIO_set：重新设置一个 BIO 对象的类型。

BIO_free、_vfree、_free_all：释放单个 BIO 的内存和资源，或释放整个 BIO 链。

BIO_push：将一个 BIO 对象加入到一个 BIO 或者 BIO 链。

BIO_pop：将一个 BIO 对象从一个 BIO 链中移走。

BIO_flush：将 BIO 内部缓冲区的数据都写出去。

BIO_read：从 BIO 读取指定字节数的数据到指定缓冲区中。

BIO_write：从数据缓冲区 out 中向 BIO 写入指定字节数的数据。

BIO_puts：将字符串 buf 写入 BIO 中。

BIO_printf：向 BIO 中按照指定格式写入数据到 BIO 中。

Source/Sink 类型 BIO 如下。

BIO_s_mem：内存缓冲区——读写 Byte 型数组数据，并能增长一直到耗尽内存。

BIO_s_file：文件指针——封装的 FILE * 对象，也可方便用于标准输入/输出。

BIO_s_fd：文件描述符—封装的文件描述符对象。

BIO_s_socket：Socket——封装了 Socket 对象。

BIO_s_null：什么也不能读写。

Filter 类型 BIO 如下。

BIO_f_buffer：I/O 缓冲，通过此 filter 的数据被缓冲到内存中。

BIO_f_md：消息摘要计算，可以用于计算通过此 filter 的数据的摘要。

BIO_f_cipher：加解密从此 filter 中经过的数据。

BIO_f_base64：base 64 编解码功能。

BIO_f_ssl：对通过此 filter 的数据执行 SSL 加密。

2. 编程要求

利用 OpenSSL 实现安全的 Web Server 的具体要求如下。

(1) 在理解 HTTPS 及 SSL 的工作原理的基础上，实现安全的 Web Server。

(2) Server 能够并发处理多个请求，要求至少能支持 Get 命令。可以增加 Web Server

的功能，如支持 Head、Post 以及 Delete 命令等。

(3) 编写必要的客户端测试程序，用户发送 HTTPS 请求并显示返回结果，也可以使用一般的 Web 浏览器程序。

3. 实现代码

Web 服务模块在类 CHttpProtocol 中实现。该类封装了 HTTPS 中与本程序相关的操作，主要包括监听线程函数以及客户端线程函数中需要调用的子函数的定义及实现。其中，比较核心的子函数包括 SSL 分析请求，将响应头部返回给客户端以及将文件发送回客户端等。具体模块实现如下。

(1) 监听线程：用于监听 Web 服务连接请求。

使用 pthread_create( )函数创建监听线程，代码如下。

```
int CHttpProtocol::TcpListen()
{
  int sock;
    struct sockaddr_in sin;
    if((sock = socket(PF_INET,SOCK_STREAM,0))< 0)
      err_exit("Couldn't make socket");

    memset(&sin,0,sizeof(sin));
    sin.sin_addr.s_addr = INADDR_ANY;
    sin.sin_family = PF_INET;
    sin.sin_port = htons(8000);
    if(bind(sock,(struct sockaddr *)&sin, sizeof(struct sockaddr))< 0)
      err_exit("Couldn't bind");
    listen(sock,5);
//printf("TcpListen Ok\n");

    return sock;
}

bool CHttpProtocol::SSLRecvRequest(SSL *ssl,BIO *io, LPBYTE pBuf, DWORD dwBufSize)
{
//printf("SSLRecvRequest \n");
char buf[BUFSIZZ];
    int r, length = 0;

memset(buf, 0, BUFSIZZ);                //初始化缓冲区
while(1)
{
     r = BIO_gets(io, buf, BUFSIZZ - 1);
     //printf("r =  %d\r\n",r);
     switch(SSL_get_error(ssl, r))
     {
          case SSL_ERROR_NONE:
               memcpy(&pBuf[length], buf, r);
               length += r;
               //printf("Case 1... \r\n");
```

```
                break;
            default:
                //printf("Case 2... \r\n");
                break;
        }
        //直到读到代表 HTTP 头部结束的空行
        if(!strcmp(buf,"\r\n") || !strcmp(buf,"\n"))
        {
            printf("IF...\r\n");
            break;
        }
    }
//添加结束符
pBuf[length] = '\0';
return true;
}
bool CHttpProtocol::StartHttpSrv()
{
CreateTypeMap();

printf(" ****************** Server starts ********************** \n");

pid_t pid;
m_listenSocket = TcpListen();

pthread_t listen_tid;
pthread_create(&listen_tid,NULL,&ListenThread,this);
}

void * CHttpProtocol::ListenThread(LPVOID param)
{
    printf("Starting ListenThread... \n");

    CHttpProtocol *pHttpProtocol = (CHttpProtocol *)param;
    SOCKET    socketClient;
    pthread_t  client_tid;
    struct sockaddr_in  SockAddr;
    PREQUEST  pReq;
    socklen_t  nLen;
    DWORD    dwRet;
while(1)          //循环等待,如有客户连接请求,则接受客户机连接要求
{
    //printf("while!\n");
    nLen = sizeof(SockAddr);
    //套接字等待连接,返回对应已接受的客户机连接的套接字
    socketClient = accept(pHttpProtocol->m_listenSocket, (LPSOCKADDR)&SockAddr, &nLen);
    //printf(" %s ",inet_ntoa(SockAddr.sin_addr));
    if (socketClient == INVALID_SOCKET)
    {
        printf("INVALID_SOCKET !\n");
```

```
            break;
        }
        pReq = new REQUEST;
        //pReq->hExit  = pHttpProtocol->m_bExit;
        pReq->Socket = socketClient;
        pReq->hFile = -1;
        pReq->dwRecv = 0;
        pReq->dwSend = 0;
        pReq->pHttpProtocol = pHttpProtocol;
        pReq->ssl_ctx = pHttpProtocol->ctx;
        //创建 client 进程,处理 request
        //printf("New request");
        pthread_create(&client_tid,NULL,&ClientThread,pReq);
    } //while
        return NULL;
    }
```

(2) 客户端线程：用于处理接收到的客户端请求。客户端线程由监听线程在 accept( ) 函数调用成功后创建。在客户端线程函数中，首先将根据之前初始化的上下文创建出来的 SSL 对象绑定在一个 Socket 类型的 BIO 上面，然后调用 SSL_accept( )函数，与客户端进行握手。

```
    void * CHttpProtocol::ClientThread(LPVOID param)
    {
    printf("Starting ClientThread... \n");
    int nRet;
    SSL * ssl;
    BYTE buf[4096];
    BIO * sbio, * io, * ssl_bio;
    PREQUEST pReq = (PREQUEST)param;
    CHttpProtocol * pHttpProtocol = (CHttpProtocol * )pReq->pHttpProtocol;
    //pHttpProtocol->CountUp();                  //记数
    SOCKET s = pReq->Socket;
        sbio = BIO_new_socket(s, BIO_NOCLOSE);   //创建一个 Socket 类型的 BIO 对象
    ssl = SSL_new(pReq->ssl_ctx);                //创建一个 SSL 对象
      SSL_set_bio(ssl, sbio, sbio);              //把 SSL 对象绑定到 Socket 类型的 BIO 上
    //连接客户端,在 SSL_accept 过程中,将会占用很大的 CPU
       nRet = SSL_accept(ssl);
    //nRet<=0 时发生错误
       if(nRet <= 0)
       {
           pHttpProtocol->err_exit("SSL_accept()error! \r\n");
           //return 0;
       }

       io = BIO_new(BIO_f_buffer());  //封装了缓冲区操作的 BIO,写入该接口的数据一般是准备传
                                      //入下一个 BIO 接口的,从该接口读出的数据一般也是从另
                                      //一个 BIO 传过来的
```

```
    ssl_bio = BIO_new(BIO_f_ssl());     //封装了OpenSSL的SSL协议的BIO类型为SSL协议
                                        //增加了一些BIO操作方法
    BIO_set_ssl(ssl_bio, ssl, BIO_CLOSE);// 把ssl(SSL对象)封装在ssl_bio(SSL_BIO对象)中
    BIO_push(io, ssl_bio);  // 把ssl_bio封装在一个缓冲的BIO对象中,这种方法允许我们使用
                            //BIO_* 函数族来操作新类型的IO对象,从而实现对SSL连接的缓
                            //冲读和写,接收request data
printf("****************\r\n");
```

(3) 接收客户端请求。

```
if (!pHttpProtocol->SSLRecvRequest(ssl,io,buf,sizeof(buf)))
{
    // 处理错误
    pHttpProtocol->err_exit("Receiving SSLRequest error!! \r\n");
}
else
{
        printf("Request received!! \n");
        printf("%s \n",buf);
}
nRet = pHttpProtocol->Analyze(pReq, buf);
if (nRet)
{
    // 处理错误
    pHttpProtocol->Disconnect(pReq);
    delete pReq;
    pHttpProtocol->err_exit("Analyzing request from client error!!\r\n");
}
// 生成并返回头部
if(!pHttpProtocol->SSLSendHeader(pReq,io))
{
    pHttpProtocol->err_exit("Sending fileheader error!\r\n");
}
BIO_flush(io);
// 向client传送数据
if(pReq->nMethod == METHOD_GET)
{
    printf("Sending.............................\n");
    if(!pHttpProtocol->SSLSendFile(pReq,io))
    {
        return 0;
    }
}
printf("File sent!!");
pHttpProtocol->Disconnect(pReq);
delete pReq;
SSL_free(ssl);
return NULL;
}
```

(4) HTTPS 协议分析。

```
int  CHttpProtocol::Analyze(PREQUEST pReq, LPBYTE pBuf)
{
//分析接收到的信息
char szSeps[] = " \n";
char * cpToken;
//防止非法请求
if (strstr((const char * )pBuf, "..") != NULL)
{
    strcpy(pReq->StatuCodeReason, HTTP_STATUS_BADREQUEST);
    return 1;
}

//判断 ruquest 的 method
cpToken = strtok((char * )pBuf, szSeps);       //缓存中字符串分解为一组标记串
if (!strcmp(cpToken, "GET"))                    //GET 命令
{
    pReq->nMethod = METHOD_GET;
}
else if (!strcmp(cpToken, "HEAD"))              //HEAD 命令
{
    pReq->nMethod = METHOD_HEAD;
}
else
{
        strcpy(pReq->StatuCodeReason, HTTP_STATUS_NOTIMPLEMENTED);
    return 1;
}
//获取 Request-URI
cpToken = strtok(NULL, szSeps);
if (cpToken == NULL)
{
    strcpy(pReq->StatuCodeReason, HTTP_STATUS_BADREQUEST);
    return 1;
}
strcpy(pReq->szFileName, m_strRootDir);
if (strlen(cpToken) > 1)
{
    strcat(pReq->szFileName, cpToken);          //把该文件名添加到结尾处形成路径
}
else
{
    strcat(pReq->szFileName, "/index.html");
}
printf("%s\r\n",pReq->szFileName);

return 0;
}
```

```
int CHttpProtocol::FileExist(PREQUEST pReq)
{
pReq->hFile = open(pReq->szFileName,O_RDONLY);
// 如果文件不存在,则返回出错信息
if (pReq->hFile == -1)
{
    strcpy(pReq->StatuCodeReason, HTTP_STATUS_NOTFOUND);
    printf("open %s error\n",pReq->szFileName);
    return 0;
}
else
{
    return 1;
}
}
void CHttpProtocol::Test(PREQUEST pReq)
{
    struct stat buf;
    long fl;
    if(stat(pReq->szFileName, &buf)<0)
    {
       err_exit("Getting filesize error!!\r\n");
    }
    fl = buf.st_size;
    printf("Filesize = %d\r\n",fl);
}

void CHttpProtocol::GetCurrentTime(LPSTR lpszString)
{
// 格林威治时间的星期转换
char *week[] = {"Sun,", "Mon,","Tue,","Wed,","Thu,","Fri,","Sat,",};
// 格林威治时间的月份转换
char *month[] = {"Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov",
"Dec",};
// 活动本地时间
struct tm *st;
long ct;
ct = time(&ct);
st = (struct tm *)localtime(&ct);
// 时间格式化
    sprintf(lpszString, "%s %02d %s %d %02d:%02d:%02d GMT",week[st->tm_wday], st->
tm_mday, month[st->tm_mon],
     1900+st->tm_year, st->tm_hour, st->tm_min, st->tm_sec);
}

bool CHttpProtocol::GetContentType(PREQUEST pReq, LPSTR type)
{
// 取得文件的类型
    char * cpToken;
```

```
    cpToken = strstr(pReq->szFileName, ".");
     strcpy(pReq->postfix, cpToken);
//遍历搜索该文件类型对应的content-type
map<char *, char *>::iterator it = m_typeMap.find(pReq->postfix);
if(it != m_typeMap.end())
{
    sprintf(type,"%s",(*it).second);
}
return 1;
}
```

(5) 发送 HTTPS 响应。

服务器端会首先根据客户端请求的命令和文件，来判断命令是否符合标准，所请求的文件上是否存在，然后组成一个响应发回客户端。人们通常在上网的时候，网页上出现 404 或 400 等错误，就是因为这一步请求的文件在服务器上不存在或者命令错误，服务器就会发送一个错误的响应到用户的浏览器上。

下面的代码摘自服务器端的响应函数，该函数由客户端线程函数调用。

```
bool CHttpProtocol::SSLSendHeader(PREQUEST pReq, BIO *io)
{
char Header[2048] = " ";
int n = FileExist(pReq);
if(!n)                    //文件不存在，则返回
{
    err_exit("The file requested doesn't exist!");
}
char curTime[50];
GetCurrentTime(curTime);
//取得文件长度
struct stat buf;
long length;
if(stat(pReq->szFileName, &buf)<0)
{
   err_exit("Getting filesize error!!\r\n");
}
length = buf.st_size;
//取得文件的类型
char ContentType[50] = " ";
GetContentType(pReq, (char *)ContentType);

sprintf((char *)Header, "HTTP/1.1 %s\r\nDate: %s\r\nServer: %s\r\nContent-Type: %s\r\
nContent-Length: %d\r\n\r\n",
                              HTTP_STATUS_OK,
                              curTime,                    //Date
                              "Villa Server 192.168.1.49", //Server"My Https Server"
                              ContentType,                //Content-Type
                              length);                    //Content-length
```

```
if(BIO_write(io, Header,strlen(Header)) <= 0)                    //错误
{
    return false;
}
BIO_flush(io);
printf("SSLSendHeader successfully!\n");
return true;
}

bool CHttpProtocol::SSLSendFile(PREQUEST pReq, BIO * io)
{
int n = FileExist(pReq);
// 如果请求的文件不存在,则返回
if(!n)
{
    err_exit("The file requested doesn't exist!");
}
static char buf[2048];
    DWORD  dwRead;
    BOOL   fRet;
int flag = 1,nReq;
    // 读写数据直到完成
    while(1)
{
    // 从 file 中读入到 buffer 中
    fRet = read(pReq->hFile, buf, sizeof(buf));
    if (fRet < 0)
    {
    static char szMsg[512];
        sprintf(szMsg, "%s", HTTP_STATUS_SERVERERROR);
// 向客户端发送出错信息
        if((nReq = BIO_write(io, szMsg,strlen(szMsg))) <= 0)     //错误
        {
            err_exit("BIO_write() error!\n");
        }
        BIO_flush(io);
    break;
    }
    if (fRet == 0)
    {
        printf("complete \n");
        break;
    }
    // 将 buffer 内容传送给 client
```

```
        if(BIO_write(io, buf, fRet) <= 0)                            //错误
        {
            if(! BIO_should_retry(io))
            {
                printf("BIO_write() error!\r\n");
                break;
            }
        }
        BIO_flush(io);
        //统计发送流量
        pReq->dwSend += fRet;
    }
    // 关闭文件
    if (close(pReq->hFile) == 0)
    {
        pReq->hFile = -1;
        return true;
    }
    else                                                             //错误
    {
        err_exit("Closing file error!");
    }
}
```

(6) 释放相关资源。

在客户端线程完成请求后,需要释放相关资源,代码如下。

```
SSL_shutdown(ssl) ;         //关闭 SSL 连接
SSL_free(ssl);              //释放 SL 结构
SSL_CTX_free(ssl_ctx);      //释放上下文环境
```

8.2 安全电子邮件编程

8.2.1 基础知识

电子邮件系统也是网络与外部必须开放的服务系统。由于电子邮件系统的复杂性,其被发现的安全漏洞非常多,并且危害很大。加强电子邮件系统的安全性,通常有如下办法。

(1) 设置一台位于停火区的电子邮件服务器作为内外电子邮件通信的中转站(或利用防火墙的电子邮件中转功能)。所有出入的电子邮件均通过该中转站中转。

(2) 同样为该服务器安装实施监控系统。

(3) 该邮件服务器作为专门的应用服务器,不运行任何其他业务(切断与内部网的通信)。

(4) 升级到最新的安全版本。

1. Internet 邮件的传输过程

电子邮件的工作过程遵循客户-服务器模式。每份电子邮件的发送都要涉及发送方与接收方，发送方式构成客户端，而接收方构成服务器，服务器含有众多用户的电子信箱。发送方通过邮件客户程序，将编辑好的电子邮件向邮局服务器(SMTP 服务器)发送。邮局服务器识别接收者的地址，并向管理该地址的邮件服务器(POP3 服务器)发送消息。邮件服务器将消息存放在接收者的电子信箱内，并告知接收者有新邮件到来。接收者通过邮件客户程序连接到服务器后，就会看到服务器的通知，进而打开自己的电子信箱来查收邮件。

通常 Internet 上的个人用户不能直接接收电子邮件，而是通过申请 ISP 主机的一个电子信箱，由 ISP 主机负责电子邮件的接收。一旦有用户的电子邮件到来，ISP 主机就将邮件移到用户的电子信箱内，并通知用户有新邮件。因此，当发送一条电子邮件给另一个客户时，电子邮件首先从用户计算机发送到 ISP 主机，再到 Internet，再到收件人的 ISP 主机，最后到收件人的个人计算机。

ISP 主机起着"邮局"的作用，管理着众多用户的电子信箱。每个用户的电子信箱实际上就是用户所申请的账户名。每个用户的电子邮件信箱都要占用 ISP 主机一定容量的硬盘空间，由于这一空间是有限的，因此用户要定期查收和阅读电子信箱中的邮件，以便腾出空间来接收新的邮件。

电子邮件在发送与接收过程中都要遵循 SMTP、POP3 等协议，这些协议确保了电子邮件在各种不同系统之间的传输。其中，SMTP 负责电子邮件的发送，而 POP3 则用于接收 Internet 上的电子邮件。

因为传递的方式不同，所以我们将内容分为两个部分来讨论：本地与远程网络邮件传递。本地网络邮件传递如果电子邮件的发件人和收件人邮箱都位于同一台邮件服务器中，它会利用以下的方式进行邮件传递，如图 8-2 所示。

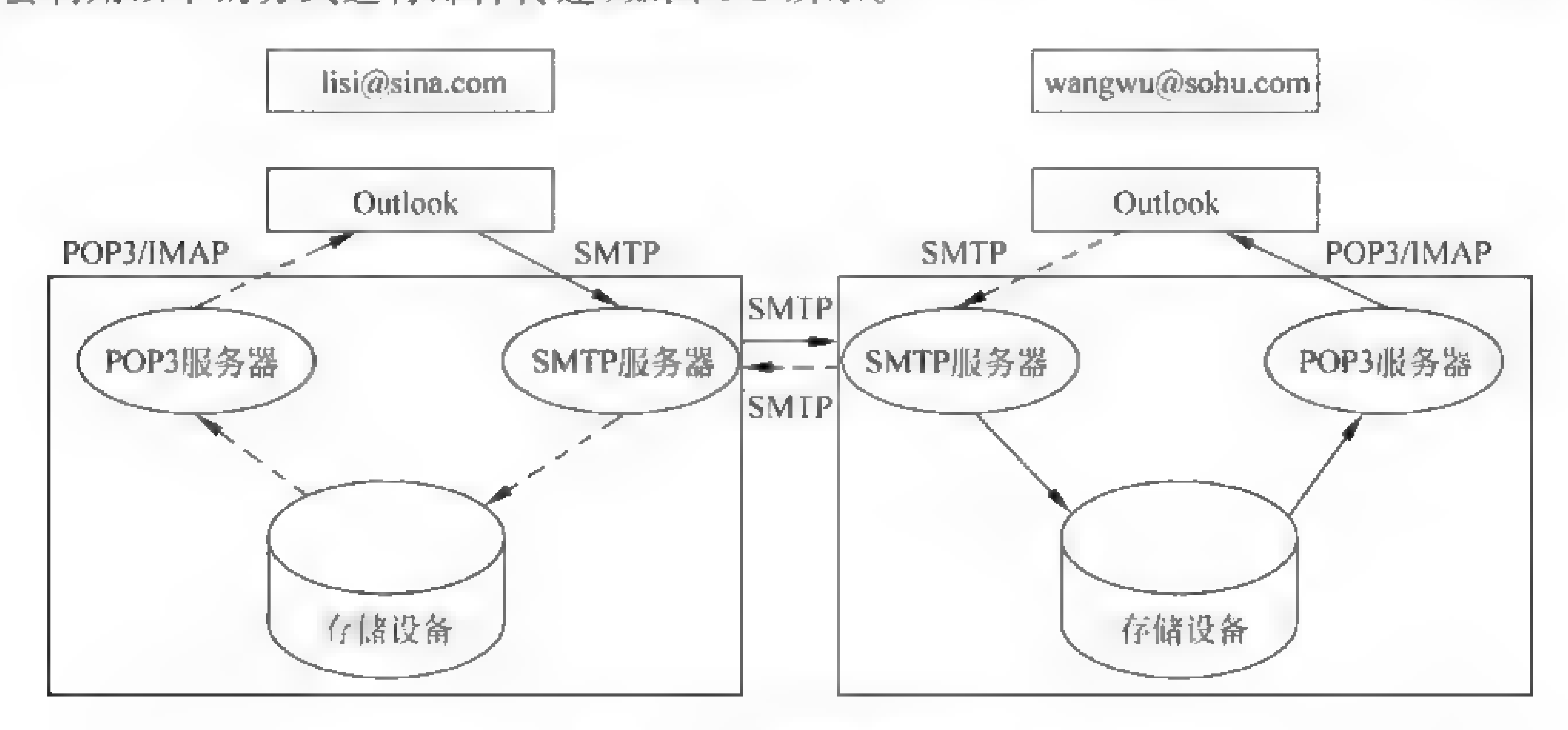


图 8-2　电子邮件传输过程

(1) MUA 先利用 TCP 连接端口 25，将电子邮件传送到邮件服务器(MTA)，此时发件人必须正确定义本身与收件人的电子邮件地址，然后这些邮件会先保存在队列中。

(2) 经过服务器的判断，如果收件人属于本地网络的用户，则此邮件就会交由 MDA 进行处理，之后直接传送到收件人邮箱。

(3) 收件人利用 POP 或 IMAP 软件(MUA),连接到邮件服务器下载或直接读取电子邮件,整个邮件传递过程也随之完成。注意:如果网络中断或拥塞,信件会一直暂存在系统的队列(/var/spool mqueue 目录),等一段时间后再尝试传送。由于发件人与收件人位于同一网络中,而且双方的电子邮件邮箱也都在同一台邮件服务器上,因此不一定需要通过主机名称或域名来查找收件人,唯一需要的是用户的账户名称,因为在同一台服务器上不会存在两个相同的账名名称。

例如,同一网络中的用户要寄一封电子邮件给另一用户 caroline,则可以使用的收件人地址类型有以下几种。

```
?caroline@mail.fc5linux.com
?caroline@mail
?caroline@localhost
?caroline@
?caroline
```

上述的第一种电子邮件地址类型是最完整的表示法,caroline 表示用户账名名称,mail 表示邮件服务器的别名,而 fc5linux.com 则是已向 InterNIC 注册的域名。

如果电子邮件的发件人和收件人位于不同的网络中,例如中国台湾和美国,它的邮件传递较为复杂,远程网络邮件传递一般的步骤如下。

(1) MUA 先利用 TCP 连接端口 25,将电子邮件传送到 MTA,此时发件人必须正确定义本身与收件人的电子邮件地址,然后这些邮件会先保存在队列中。

(2) 经过服务器的判断,如果收件人属于远程网络的用户,则此服务器会先向 DNS 服务器要求解析远程邮件服务器的 IP 地址。

(3) 如果名称解析失败,则无法进行邮件的传递。如果成功解析远程邮件服务器的 IP 地址,则本地的邮件服务器将利用 SMTP 将邮件传送到远程(这就是邮件转发功能)。

(4) SMTP 将尝试和远程的邮件服务器连接,如果远程服务器目前无法接收邮件,则这些信件会继续停留在队列中,然后在指定的重试间隔再次尝试连接,直到成功或放弃传送为止。

(5) 如果传送成功,则远程 MTA 就会将此邮件交由 MDA 进行处理,并放入用户邮箱。之后收件人即可利用 POP 或 IMAP 软件,连接到邮件服务器下载或读取电子邮件,而整个邮件传递过程也随之完成。

综合以上两种不同形式的电子邮件传递方式,图 8-3 是完整的电子邮件传递流程。

2. SMTP

SMTP 是一种 TCP 支持的提供可靠且有效电子邮件传输的应用层协议。SMTP 是建立在 TCP 上的一种邮件服务,主要用于传输系统之间的邮件信息并提供与来信有关的通知。

SMTP 独立于特定的传输子系统,且只需要可靠有序的数据流信道支持。SMTP 的重要特性之一是其能跨越网络传输邮件,即"SMTP 邮件中继"。通常,一个网络可以由公用互联网上 TCP 可相互访问的主机、防火墙分隔的 TCP/IP 网络上 TCP 可相互访问的主机,及其他 LAN/WAN 中的主机利用非 TCP 传输层协议组成。使用 SMTP,可实现相同网络

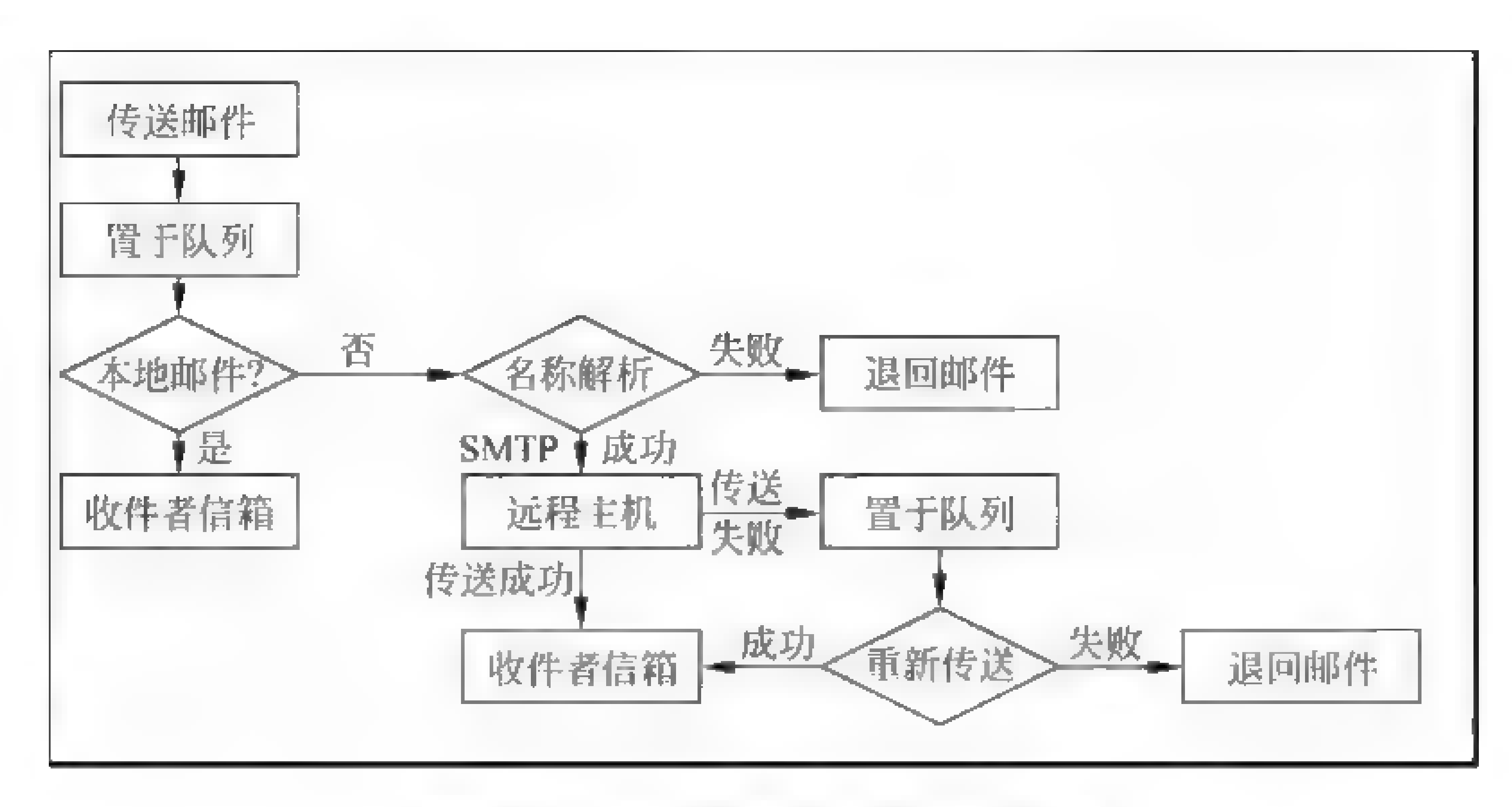


图 8-3 完整的电子邮件传递过程

上处理机之间的邮件传输，也可通过中继器或网关实现某处理机与其他网络之间的邮件传输。在这种方式下，邮件的发送可能经过从发送端到接收端路径上的大量中间中继器或网关主机。域名服务系统(DNS)的邮件交换服务器可以用来识别出传输邮件的下一条 IP 地址。

SMTP 是一个相对简单的基于文本的协议。在其之上指定了一条消息的一个或多个接收者(在大多数情况下被确认是存在的)，然后消息文本会被传输。可以很简单地通过 Telnet 程序来测试一个 SMTP 服务器。SMTP 使用 TCP 端口 25。要为一个给定的域名决定一个 SMTP 服务器，需要使用 MX (Mail eXchange)DNS。

在 20 世纪 80 年代早期，SMTP 开始被广泛地使用。当时，它只是作为 UUCP 的补充，UUCP 更适合于处理在间歇连接的机器间传送邮件。相反，SMTP 在发送和接收的机器始终连接在网络的情况下工作得最好。Sendmail 是最早实现 SMTP 的邮件传输代理之一。到 2001 年至少有 50 个程序将 SMTP 实现为一个客户端(消息的发送者)或一个服务器(消息的接收者)。一些其他的流行的 SMTP 服务器程序包括 Philip Hazel 的 exim，IBM 的 Postfix，D. J. Bernstein 的 Qmail，以及 Microsoft Exchange Server。由于这个协议开始是基于纯 ASCII 文本的，它在二进制文件上处理得并不好。诸如 MIME 的标准被开发出来编码二进制文件以使其通过 SMTP 来传输。今天，大多数 SMTP 服务器都支持 8 位 MIME 扩展，它使二进制文件的传输变得几乎和纯文本一样简单。

SMTP 是一个“推”的协议，它不允许根据需要从远程服务器上“拉”来消息。要做到这点，邮件客户端必须使用 POP3 或 IMAP。另一个 SMTP 服务器可以使用 ETRN 在 SMTP 上触发一个发送。

简单邮件传输协议(SMTP)是一种基于文本的电子邮件传输协议，是在因特网中用于在邮件服务器之间交换邮件的协议。SMTP 是应用层的服务，可以适应于各种网络系统。

SMTP 的命令和响应都是基于文本，以命令行为单位，换行符为 CR LF。响应信息一般只有一行，由一个三位数的代码开始，后面可附上很简短的文字说明。

SMTP 要经过建立连接、传送邮件和释放连接三个阶段，具体如下。

(1) 建立 TCP 连接。

(2) 客户端向服务器发送 HELO 命令以标识发件人自己的身份，然后客户端发送

MAIL命令。

(3) 服务器端以OK作为响应,表示准备接收。

(4) 客户端发送RCPT命令。

(5) 服务器端表示是否愿意为收件人接收邮件。

(6) 协商结束,发送邮件,用命令DATA发送输入内容。

(7) 结束此次发送,用QUIT命令退出。

SMTP服务器基于DNS中的邮件交换(MX)记录路由电子邮件。电子邮件系统发邮件时是根据收信人的地址后缀来定位邮件服务器的。SMTP通过用户代理程序(UA)完成邮件的编辑、收取和阅读等功能;通过邮件传输代理程序(MTA)将邮件传送到目的地。

3. POP3与IMAP

POP3,全名为Post Office Protocol-Version 3,即"邮局协议版本3"。它是TCP/IP协议族中的一员,由RFC 1939定义。本协议主要用于支持使用客户端远程管理在服务器上的电子邮件。提供了SSL加密的POP3协议被称为POP3S。

POP协议支持"离线"邮件处理。其具体过程是:邮件发送到服务器上,电子邮件客户端调用邮件客户机程序以连接服务器,并下载所有未阅读的电子邮件。这种离线访问模式是一种存储转发服务,将邮件从邮件服务器端送到个人终端机器上,一般是PC或MAC。一旦邮件发送到PC或MAC上,邮件服务器上的邮件将会被删除。但目前的POP3邮件服务器大都可以"只下载邮件,服务器端并不删除",也就是改进的POP3协议。

POP3协议允许电子邮件客户端下载服务器上的邮件,但是在客户端的操作(如移动邮件、标记已读等),不会反馈到服务器上,比如通过客户端收取了邮箱中的三封邮件并移动到其他文件夹,邮箱服务器上的这些邮件是没有同时被移动的。

而IMAP提供Webmail与电子邮件客户端之间的双向通信,客户端的操作都会反馈到服务器上,对邮件进行的操作,服务器上的邮件也会做相应的动作。

同时,IMAP像POP3那样提供了方便的邮件下载服务,让用户能进行离线阅读。IMAP提供的摘要浏览功能可以让用户在阅读完所有的邮件到达时间、主题、发件人、大小等信息后才做出是否下载的决定。此外,IMAP更好地支持了从多个不同设备中随时访问新邮件。

总之,IMAP整体上为用户带来更为便捷和可靠的体验。POP3更易丢失邮件或多次下载相同的邮件,但IMAP通过邮件客户端与Webmail之间的双向同步功能很好地避免了这些问题。

若在Web邮箱中设置了"保存到已发送",使用客户端POP服务发信时,已发邮件也会自动同步到网页端"已发送"文件夹内。

8.2.2 编程训练——实现安全电子邮件传输

编程训练目的:通过实验设计在发送邮件客户端利用DES加密算法对邮件进行加密,在接收邮件客户端对邮件进行解密,从而保证电子邮件在网络中的安全传输。通过对本实验中的安全电子邮件的设计和开发,加深对与电子邮件相关的协议的理解,熟悉网络客户端与网络服务器之间的通信机制,熟悉利用加密算法对电子邮件进行加密的原理。

1. 程序的需求

安全电子邮件必须要完成以下几个主要的功能。

(1) 撰写：给用户提供方便的编辑信件的环境。

(2) 显示：能方便地在计算机上显示用户的来信，以使用户方便地阅读和存储。

(3) 处理：包括发送邮件和接收邮件。

(4) 加密：安全电子邮件代理能够对发送的邮件进行加密从而保证邮件在网络中的安全传输。

(5) 认证：确保用户收到的邮件内容没有被篡改。

2. 邮件发送接收流程

根据以上需求，使用 POP3 和 SMTP 实现了一个接收邮件的程序和一个发送邮件的程序。发送邮件和接收邮件流程分别如图 8-4 和图 8-5 所示。

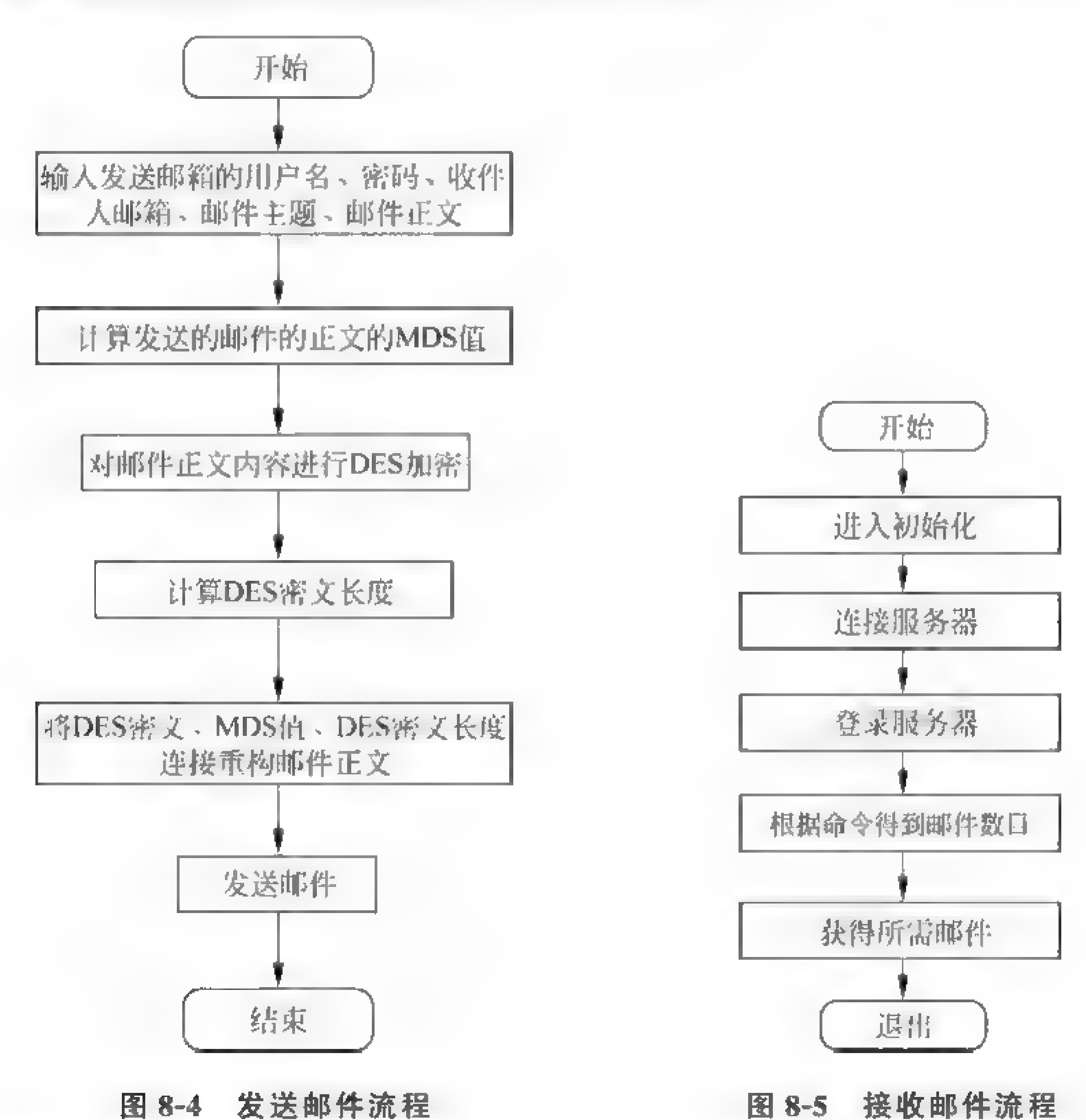


图 8-4 发送邮件流程　　图 8-5 接收邮件流程

3. 安全邮件传输代码实现

1) 发送邮件

```
/*smtp.h*/
#ifndef __SMTP_H__                    //避免重复包含
#define __SMTP_H__
#include<iostream>
#include<list>
#include<string>
```

```
#include <WinSock2.h>
using namespace std;
const int MAXLEN = 1024;
const int MAX_FILE_LEN = 6000;
static const char base64Char[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz01234
56789+/";                        //base64 编码
struct FILEINFO                  /*用来记录文件的一些信息*/
{
char fileName[128];              /*文件名称*/
char filePath[256];              /*文件绝对路径*/
};

class CSmtp
{
public:
CSmtp(void);
//构造函数
CSmtp(
    int port,
    string srvDomain,            //SMTP 服务器域名
    string userName,             //用户名
    string password,             //密码
    string targetEmail,          //目的邮件地址
    string emailTitle,           //主题
    string content               //内容
    );
public:
~CSmtp(void);
public:
int port;
public:
string domain;
string user;
string pass;
string targetAddr;
string title;
string content;

list<FILEINFO *> listFile;
public:
char buff[MAXLEN + 1];
int buffLen;
SOCKET sockClient;               //客户端的套接字
public:
bool CreateConn();               /*创建连接*/

bool Send(string &message);
bool Recv();

void FormatEmailHead(string &email);            //格式化要发送的邮件头部
```

```
int Login();
bool SendEmailHead();                          //发送邮件头部信息
bool SendTextBody();                           //发送文本信息
//bool SendAttachment();                       //发送附件
int SendAttachment_Ex();
bool SendEnd();
public:
void AddAttachment(string &filePath);          //添加附件
void DeleteAttachment(string &filePath);       //删除附件
void DeleteAllAttachment();                    //删除所有的附件

void SetSrvDomain(string &domain);
void SetUserName(string &user);
void SetPass(string &pass);
void SetTargetEmail(string &targetAddr);
void SetEmailTitle(string &title);
void SetContent(string &content);
void SetPort(int port);
int SendEmail_Ex();
/* 关于错误码的说明:1.网络错误导致的错误 2.用户名错误 3.密码错误 4.文件不存在 0.成功 */
char* base64Encode(char const* origSigned, unsigned origLength);
};

#endif // !__SMTP_H__


/* smtp.cpp */
#include "smtp.h"
#include <iostream>
#include <fstream>
using namespace std;

#pragma  comment(lib, "ws2_32.lib")            /* 连接 ws2_32.lib 动态链接库 */

char* CSmtp::base64Encode(char const* origSigned, unsigned origLength)
{
unsigned char const* orig = (unsigned char const*)origSigned;
                                               //in case any input bytes have the MSB set
if (orig == NULL) return NULL;

unsigned const numOrig24BitValues = origLength / 3;
bool havePadding = origLength > numOrig24BitValues * 3;
bool havePadding2 = origLength == numOrig24BitValues * 3 + 2;
unsigned const numResultBytes = 4 * (numOrig24BitValues + havePadding);
char* result = new char[numResultBytes + 3];   //allow for trailing '/0'

//Map each full group of 3 input bytes into 4 output base-64 characters:
unsigned i;
for (i = 0; i < numOrig24BitValues; ++i)
{
```

```
        result[4 * i + 0] = base64Char[(orig[3 * i] >> 2) & 0x3F];
        result[4 * i + 1] = base64Char[(((orig[3 * i] & 0x3) << 4) | (orig[3 * i + 1] >> 4)) & 0x3F];
        result[4 * i + 2] = base64Char[((orig[3 * i + 1] << 2) | (orig[3 * i + 2] >> 6)) & 0x3F];
        result[4 * i + 3] = base64Char[orig[3 * i + 2] & 0x3F];
    }

    //Now, take padding into account.  (Note: i == numOrig24BitValues)
    if (havePadding)
    {
        result[4 * i + 0] = base64Char[(orig[3 * i] >> 2) & 0x3F];
        if (havePadding2)
        {
            result[4 * i + 1] = base64Char[(((orig[3 * i] & 0x3) << 4) | (orig[3 * i + 1] >> 4))
    & 0x3F];
            result[4 * i + 2] = base64Char[(orig[3 * i + 1] << 2) & 0x3F];
        }
        else
        {
            result[4 * i + 1] = base64Char[((orig[3 * i] & 0x3) << 4) & 0x3F];
            result[4 * i + 2] = '=';
        }
        result[4 * i + 3] = '=';
    }

    result[numResultBytes] = '\0';
    return result;
    }
    CSmtp::CSmtp(void)
    {
    this->content = "";
    this->port = 25;
    this->user = "";
    this->pass = "";
    this->targetAddr = "";
    this->title = "";
    this->domain = "";

    WORD wVersionRequested;
    WSADATA wsaData;
    int err;
    wVersionRequested = MAKEWORD(2, 1);
    err = WSAStartup(wVersionRequested, &wsaData);
    this->sockClient = 0;
    }

    CSmtp::~CSmtp(void)
    {
    DeleteAllAttachment();
    closesocket(sockClient);
    WSACleanup();
```

```
}
CSmtp::CSmtp(
int port,
string srvDomain,
string userName,
string password,
string targetEmail,
string emailTitle,
string content
)
{
this->content = content;
this->port = port;
this->user = userName;
this->pass = password;
this->targetAddr = targetEmail;
this->title = emailTitle;
this->domain = srvDomain;

WORD wVersionRequested;
WSADATA wsaData;
int err;
wVersionRequested = MAKEWORD(2, 1);
err = WSAStartup(wVersionRequested, &wsaData);
this->sockClient = 0;
}

bool CSmtp::CreateConn()
{
//为建立 socket 对象做准备,初始化环境
SOCKET sockClient = socket(AF_INET, SOCK_STREAM, 0);        //建立 socket 对象
SOCKADDR_IN addrSrv;
HOSTENT* pHostent;
pHostent = gethostbyname(domain.c_str());                  //得到有关于域名的信息

addrSrv.sin_addr.S_un.S_addr = *((DWORD *)pHostent->h_addr_list[0]);   //得到 SMTP 服务
                                                                        //器的网络字节
                                                                        //序的 IP 地址
addrSrv.sin_family = AF_INET;
addrSrv.sin_port = htons(port);
int err = connect(sockClient, (SOCKADDR*)&addrSrv, sizeof(SOCKADDR)); //向服务器发送请求
if (err != 0)
{
    return false;
    printf("链接失败\n");
}
this->sockClient = sockClient;
if (false == Recv())
{
    return false;
```

```
}
return true;
}

bool CSmtp::Send(string &message)
{
int err = send(sockClient, message.c_str(), message.length(), 0);
if (err == SOCKET_ERROR)
{
    return false;
}
string message01;
cout << message.c_str() << endl;
return true;
}

bool CSmtp::Recv()
{
memset(buff, 0, sizeof(char) * (MAXLEN + 1));
int err = recv(sockClient, buff, MAXLEN, 0);          //接收数据
if (err == SOCKET_ERROR)
{
    return false;
}
buff[err] = '\0';
cout << buff << endl;
return true;
}

int CSmtp::Login()
{
string sendBuff;
sendBuff = "EHLO ";
sendBuff += user;
sendBuff += "\r\n";

if (false == Send(sendBuff) || false == Recv())    //既接收也发送
{
    return 1;                                       /*1 表示发送失败由于网络错误*/
}
sendBuff.empty();
sendBuff = "AUTH LOGIN\r\n";
if (false == Send(sendBuff) || false == Recv())    //请求登录
{
    return 1;
}
sendBuff.empty();
int pos = user.find('@', 0);
sendBuff = user.substr(0, pos);                     //得到用户名
char * ecode;
```

```
ecode = base64Encode(sendBuff.c_str(), strlen(sendBuff.c_str()));
sendBuff.empty();
sendBuff = ecode;
sendBuff += "\r\n";
delete[]ecode;

if (false == Send(sendBuff) || false == Recv()) //发送用户名,并接收服务器的返回
{
    return 1;                               /* 错误码 1 表示发送失败由于网络错误 */
}
sendBuff.empty();
ecode = base64Encode(pass.c_str(), strlen(pass.c_str()));
sendBuff = ecode;
sendBuff += "\r\n";
delete[]ecode;
if (false == Send(sendBuff) || false == Recv()) //发送用户密码,并接收服务器的返回
{
    return 1;
}

if (NULL != strstr(buff, "550"))
{
    return 2;                               /* 错误码 2 表示用户名错误 */
}

if (NULL != strstr(buff, "535"))            /* 535 是认证失败的返回 */
{
    return 3;                               /* 错误码 3 表示密码错误 */
}
return 0;
}

bool CSmtp::SendEmailHead()                 //发送邮件头部信息
{
string sendBuff;
sendBuff = "MAIL FROM: <" + user + ">\r\n";
if (false == Send(sendBuff) || false == Recv())
{
    return false;                           /* 表示发送失败由于网络错误 */
}
sendBuff.empty();
sendBuff = "RCPT TO: <" + targetAddr + ">\r\n";
if (false == Send(sendBuff) || false == Recv())
{
    return false;
}

sendBuff.empty();
sendBuff = "DATA\r\n";
```

```
if (false == Send(sendBuff) || false == Recv())
{
    return false;                          //表示发送失败由于网络错误
}

sendBuff.empty();
FormatEmailHead(sendBuff);
if (false == Send(sendBuff))
    //发送完头部之后不必调用接收函数,因为没有\r\n.\r\n结尾,服务器认为没有发完数据,所
    //以不会返回什么值
{
    return false;
}
return true;
}

void CSmtp::FormatEmailHead(string &email)
{                                          /*格式化要发送的内容*/
email = "From: ";
email += user;
email += "\r\n";

email += "To: ";
email += targetAddr;
email += "\r\n";
email += "Subject: ";
email += title;
email += "\r\n";
email += "MIME-Version: 1.0";
email += "\r\n";
email += "Content-Type: multipart/mixed;boundary=qwertyuiop";
email += "\r\n";
email += "\r\n";
}

bool CSmtp::SendTextBody()                 /*发送邮件文本*/
{
string sendBuff;
sendBuff = "--qwertyuiop\r\n";
sendBuff += "Content-Type: text/plain;";
sendBuff += "charset=\"gb2312\"\r\n\r\n";
sendBuff += content;
sendBuff += "\r\n\r\n";
return Send(sendBuff);
}

int CSmtp::SendAttachment_Ex()             /*发送附件*/
{
for (list<FILEINFO *>::iterator pIter = listFile.begin(); pIter != listFile.end(); pIter++)
{
```

```
        cout << "Attachment is sending ～～～～～" << endl;
        cout << "Please be patient!" << endl;
        string sendBuff;
        sendBuff = "--qwertyuiop\r\n";
        sendBuff += "Content-Type: application/octet-stream;\r\n";
        sendBuff += " name=\"";
        sendBuff += (*pIter)->fileName;
        sendBuff += "\"";
        sendBuff += "\r\n";

        sendBuff += "Content-Transfer-Encoding: base64\r\n";
        sendBuff += "Content-Disposition: attachment;\r\n";
        sendBuff += " filename=\"";
        sendBuff += (*pIter)->fileName;
        sendBuff += "\"";
        sendBuff += "\r\n";
        sendBuff += "\r\n";
        Send(sendBuff);
        ifstream ifs((*pIter)->filePath, ios::in | ios::binary);
        if (false == ifs.is_open())
        {
            return 4;                     /* 错误码 4 表示文件打开错误 */
        }
        char fileBuff[MAX_FILE_LEN];
        char *chSendBuff;
        memset(fileBuff, 0, sizeof(fileBuff));
        /* 文件使用 base64 加密传送 */
        while (ifs.read(fileBuff, MAX_FILE_LEN))
        {
            //cout << ifs.gcount() << endl;
            chSendBuff = base64Encode(fileBuff, MAX_FILE_LEN);
            chSendBuff[strlen(chSendBuff)] = '\r';
            chSendBuff[strlen(chSendBuff)] = '\n';
            send(sockClient, chSendBuff, strlen(chSendBuff), 0);
            delete[]chSendBuff;
        }
        //cout << ifs.gcount() << endl;
        chSendBuff = base64Encode(fileBuff, ifs.gcount());
        chSendBuff[strlen(chSendBuff)] = '\r';
        chSendBuff[strlen(chSendBuff)] = '\n';
        int err = send(sockClient, chSendBuff, strlen(chSendBuff), 0);

        if (err != strlen(chSendBuff))
        {
            cout << "文件传送出错!" << endl;
            return 1;
        }
        delete[]chSendBuff;
    }
    return 0;
```

```
}

bool CSmtp::SendEnd()                          /* 发送结尾信息 */
{
string sendBuff;
sendBuff = "--qwertyuiop--";
sendBuff += "\r\n.\r\n";
if (false == Send(sendBuff) || false == Recv())
{
    return false;
}
cout << buff << endl;
sendBuff.empty();
sendBuff = "QUIT\r\n";
return (Send(sendBuff) && Recv());
}

int CSmtp::SendEmail_Ex()
{
if (false == CreateConn())
{
    return 1;
}
//Recv();
int err = Login();                             //先登录
if (err != 0)
{
    return err;                                //错误代码必须要返回
}
if (false == SendEmailHead())                  //发送 EMAIL 头部信息
{
    return 1;                                  /* 错误码 1 是由于网络的错误 */
}
if (false == SendTextBody())
{
    return 1;
}
err = SendAttachment_Ex();
if (err != 0)
{
    return err;
}
if (false == SendEnd())
{
    return 1;
}
return 0;                                      /* 0 表示没有出错 */
}

void CSmtp::AddAttachment(string &filePath)    //添加附件
```

```
{
FILEINFO * pFile = new FILEINFO;
strcpy(pFile->filePath, filePath.c_str());
const char * p = filePath.c_str();
strcpy(pFile->fileName, p + filePath.find_last_of("\\") + 1);
listFile.push_back(pFile);
}

void CSmtp::DeleteAttachment(string &filePath)     //删除附件
{
list<FILEINFO *>::iterator pIter;
for (pIter = listFile.begin(); pIter != listFile.end(); pIter++)
{
    if (strcmp((*pIter)->filePath, filePath.c_str()) == 0)
    {
        FILEINFO * p = *pIter;
        listFile.remove(*pIter);
        delete p;
        break;
    }
}
}

void CSmtp::DeleteAllAttachment()                    /* 删除所有的文件 */
{
for (list<FILEINFO *>::iterator pIter = listFile.begin(); pIter != listFile.end();)
{
    FILEINFO * p = *pIter;
    pIter = listFile.erase(pIter);
    delete p;
}
}

void CSmtp::SetSrvDomain(string &domain)
{
this->domain = domain;
}

void CSmtp::SetUserName(string &user)
{
this->user = user;
}

void CSmtp::SetPass(string &pass)
{
this->pass = pass;
}
void CSmtp::SetTargetEmail(string &targetAddr)
{
this->targetAddr = targetAddr;
```

```
}
void CSmtp::SetEmailTitle(string &title)
{
this->title = title;
}
void CSmtp::SetContent(string &content)
{
this->content = content;
}
void CSmtp::SetPort(int port)
{
this->port = port;
}

/* main.cpp */
#include<stdlib.h>
#include "smtp.h"
#include <iostream>
#include<stdio.h>
#include<string>
#include<string.h>
#include<conio.h>
#include <openssl/md5.h>
#include <openssl/aes.h>
#include <openssl/evp.h>
#define EVP_DES_CBC EVP_des_cbc()
#define MAX_CHAR_SIZE 204800
using namespace std;

unsigned char *encrypt_text(unsigned char *iv, unsigned char *key, unsigned char *
plaintext, int *ciphertext_len, unsigned char *ciphertext)
{

EVP_CIPHER_CTX en;
EVP_CIPHER_CTX_init(&en);
const EVP_CIPHER *cipher_type;
int input_len = 0;

cipher_type = EVP_DES_CBC;

//init cipher
EVP_EncryptInit_ex(&en, cipher_type, NULL, key, iv);

const char* p = (const char*)(char*)plaintext;          //转换

input_len = strlen(p);
/* allows reusing of 'e' for multiple encryption cycles */

if (!EVP_EncryptInit_ex(&en, NULL, NULL, NULL, NULL)){
    printf("ERROR in EVP_EncryptInit_ex \n");
```

```
    return NULL;
}

int bytes_written = 0;
//encrypt
if (!EVP_EncryptUpdate(&en,
    ciphertext, &bytes_written,
    (unsigned char * )plaintext, input_len)) {
    return NULL;
}
* ciphertext_len += bytes_written;

//do padding
if (!EVP_EncryptFinal_ex(&en,
    ciphertext + bytes_written,
    &bytes_written)){
    printf("ERROR in EVP_EncryptFinal_ex \n");
    return NULL;
}
* ciphertext_len += bytes_written;

EVP_CIPHER_CTX_cleanup(&en);

return ciphertext;
}

char * MD5(string str)                              //计算 MD5 的值
{
char data[204800];
strcpy(data, str.c_str());
unsigned char md[16] = { 0 };

MD5_CTX ctx;
MD5_Init(&ctx);
MD5_Update(&ctx, data, strlen(data));
MD5_Final(md, &ctx);
int i = 0;
char buf[33] = { 0 };
char tmp[3] = { 0 };
for (i = 0; i < 16; i++)
{
    sprintf(tmp, " % 02X", md[i]);
    strcat(buf, tmp);
}
return buf;

}

void convert(char * target, char * source, int n)         //转换定长字符
{

int a;
char fstr[20];
```

```
a = atoi(source);
sprintf(fstr, "%%0%dd", n);
sprintf(target, fstr, a);
}

int main()
{
string l1, l2, l3, l4, l5, l6;
cout << "--------------------发送邮件-------------------------" << endl;
cout << "请输入发送邮件使用的用户名:";
cin >> l1;

cout << "请输入用户名密码:";
char ch;
while ((ch = _getch()) != 13)
{
    l2 += ch;                         //string 对象重载了 +=
    cout << "*";
}
cout << endl;
cout << "请输入收件人邮箱:";
cin >> l3;
cout << "请输入邮件主题:";
cin >> l4;
cout << "请输入邮件正文:";
cin >> l5;
//账号信息
l6 = "smtp.163.com";
string MD5val = MD5(l5);
//加密信息
char *m = (char*)l5.c_str();
unsigned char *in = (unsigned char*)m;
unsigned char * out = NULL;
char * iv = "wereachsuccessalready";      //初始化向量
unsigned char *i = (unsigned char*)iv;
char * key = "wearethemoststronger";      //密钥
unsigned char *k = (unsigned char*)key;
int ciphertext_len = 0;
unsigned char ciphertext[MAX_CHAR_SIZE];
unsigned char plaintext[MAX_CHAR_SIZE];
out = encrypt_text(i, k, in, &ciphertext_len, ciphertext);

out[ciphertext_len] = '\0';
l5 = (char*)out;

int l5len;                            //输入的消息内容变为密文后的长度
l5len = l5.length();
l5 += MD5val;                         //把输入的消息的 MD5 的值添加到发送的 content 中

char * l5temp = new char[];
_itoa(l5len, l5temp, 10);
convert(l5temp, l5temp, 3);
```

```
    l5 += l5temp;                    //把 content 长度添加到发送内容的后面

    CSmtp smtp(
        25,                          /* smtp 端口 */
        l6,                          /* smtp 服务器地址 */
        l1,                          /* 你的邮箱地址 */
        l2,                          /* 邮箱密码 */
        l3,                          /* 目的邮箱地址 */
        l4,                          /* 主题 */
        l5                           /* 邮件正文 */
        );

    int err;
    if ((err = smtp.SendEmail_Ex()) != 0)
    {
        if (err == 1)
            cout << "错误 1: 由于网络不畅通,发送失败!" << endl;
        if (err == 2)
            cout << "错误 2: 用户名错误,请核对!" << endl;
        if (err == 3)
            cout << "错误 3: 用户密码错误,请核对!" << endl;
        if (err == 4)
            cout << "错误 4: 请检查附件目录是否正确,以及文件是否存在!" << endl;
    }
    system("pause");
    return 0;
    return 0;
    }
```

2）接收邮件

```
/* pop3.h */
#include <WinSock2.h>
#pragma  comment(lib, "ws2_32.lib")        /* 连接 ws2_32.lib 动态链接库 */

class CPop3 {

public:
CPop3();
~CPop3();

//初始化 POP3 的属性
bool Create(const char * username, const char * userpwd, const char * svraddr,
    unsigned short port = 110);

//连接 POP3 服务器
bool Connect();
//登录的服务器
bool Login();
//利用 list 命令得到所有的邮件数目
bool List(int& sum);
//获得序号为 num 的邮件
```

```
bool FetchEx(int num = 1);
//退出命令
bool Quit();

protected:
int GetMailSum(char * buf);

SOCKET m_sock;
char m_username[32];                    /* 用户名 */
char m_userpwd[32];                     /* 密码 */
char m_svraddr[32];                     /* 服务器域名 */
unsigned short m_port;
private:
int Pop3Recv(char * buf, int len, int flags = 0);
};

/* pop3.cpp */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include "pop3.h"
#include <openssl/md5.h>
#include <openssl/aes.h>
#include <openssl/evp.h>
#define EVP_DES_CBC EVP_des_cbc()
#define MAX_CHAR_SIZE 204800
using namespace std;
//初始化
unsigned char * decrypt_text(unsigned char * iv, unsigned char * key, unsigned char *
ciphertext, int * ciphertext_len, unsigned char * plaintext) {

EVP_CIPHER_CTX de;
EVP_CIPHER_CTX_init(&de);
const EVP_CIPHER * cipher_type;

int bytes_written = 0;
int update_len = 0;
cipher_type = EVP_DES_CBC;
EVP_DecryptInit_ex(&de, cipher_type, NULL, key, iv);

if (!EVP_DecryptInit_ex(&de, NULL, NULL, NULL, NULL)){
    printf("ERROR in EVP_DecryptInit_ex \n");
    return NULL;
}
int plaintext_len = 0;
if (!EVP_DecryptUpdate(&de,
    plaintext, &update_len,
    ciphertext, *ciphertext_len)){
    printf("ERROR in EVP_DecryptUpdate\n");
```

```
        return NULL;
    }

    if (!EVP_DecryptFinal_ex(&de,
        plaintext + update_len, &bytes_written)){
        printf("ERROR in EVP_DecryptFinal_ex\n");
        return NULL;
    }
    bytes_written += update_len;
    *(plaintext + bytes_written) - '\0';

    EVP_CIPHER_CTX_cleanup(&de);

    return plaintext;
    }

    char *MD5(string str)
    {
    char data[204800];
    strcpy(data, str.c_str());
    unsigned char md[16] = { 0 };

    MD5_CTX ctx;
    MD5_Init(&ctx);
    MD5_Update(&ctx, data, strlen(data));
    MD5_Final(md, &ctx);
    int i = 0;
    char buf[33] = { 0 };
    char tmp[3] = { 0 };
    for (i = 0; i < 16; i++)
    {
        sprintf(tmp, "%02X", md[i]);
        strcat(buf, tmp);
    }
    return buf;
    }

    CPop3::CPop3()
    {
    WSADATA wsaData;
    WORD version = MAKEWORD(2, 0);
    WSAStartup(version, &wsaData);
    }

    CPop3::~CPop3()
    {
    WSACleanup();
    }
```

```
int CPop3::Pop3Recv(char * buf, int len, int flags)
{                                                    /* 接收数据 */
int rs;
int offset = 0;
do
{
    if (offset > len - 2)
        return offset;

    rs = recv(m_sock, buf + offset, len - offset, flags);
    if (rs < 0) return -1;
    offset += rs;
    buf[offset] = '\0';
} while (strstr(buf, "\r\n.\r\n") == (char *)NULL);

return offset;
}

bool CPop3::Create(
const char * username,                               //用户名
const char * userpwd,                                //用户密码
const char * svraddr,                                //服务器地址
unsigned short port                                  //服务端口
)
{
strcpy(m_username, username);
strcpy(m_userpwd, userpwd);
strcpy(m_svraddr, svraddr);
m_port = port;

return true;
}

bool CPop3::Connect()
{
//创建套接字
m_sock = socket(AF_INET, SOCK_STREAM, 0);
//IP 地址
char ipaddr[16];

struct hostent * p;
if ((p = gethostbyname(m_svraddr)) == NULL)    //如果得不到服务器信息,就说明出错
{
    return FALSE;
}

sprintf(
    ipaddr,
    "%u.%u.%u.%u",
    (unsigned char)p->h_addr_list[0][0],
```

```
        (unsigned char)p->h_addr_list[0][1],
        (unsigned char)p->h_addr_list[0][2],
        (unsigned char)p->h_addr_list[0][3]
        );

    //连接 POP 服务器
    struct sockaddr_in svraddr;
    svraddr.sin_family = AF_INET;
    svraddr.sin_addr.s_addr = inet_addr(ipaddr);
    svraddr.sin_port = htons(m_port);
    int ret = connect(m_sock, (struct sockaddr *)&svraddr, sizeof(svraddr));
    if (ret == SOCKET_ERROR)
    {
        return FALSE;
    }

    //接收 POP3 服务器发来的欢迎信息
    char buf[128];
    int rs = recv(m_sock, buf, sizeof(buf), 0);
    buf[rs] = '\0';

    printf("%s", buf);
    if (rs <= 0 || strncmp(buf, "+OK", 3) != 0)         /* 服务器没有返回 OK 就出错了 */
    {
        return FALSE;
    }

    return TRUE;
    }

    bool CPop3::Login()
    {                                                    /* 登录 */

    /* 发送用户命令 */
    char sendbuf[128];
    char recvbuf[128];

    sprintf(sendbuf, "USER %s\r\n", m_username);
    printf("%s", sendbuf);
    send(m_sock, sendbuf, strlen(sendbuf), 0);          //发送用户名

    int rs = recv(m_sock, recvbuf, sizeof(recvbuf), 0); //接收服务器发来的信息
    recvbuf[rs] = '\0';
    if (rs <= 0 || strncmp(recvbuf, "+OK", 3) != 0)     //如果没有"+OK"就说明失败了
    {
        return FALSE;
    }
    printf("%s", recvbuf);
    /* 发送密码信息 */
    memset(sendbuf, 0, sizeof(sendbuf));
```

```
sprintf(sendbuf, "PASS %s\r\n", m_userpwd);
send(m_sock, sendbuf, strlen(sendbuf), 0);
printf("%s", sendbuf);

rs = recv(m_sock, recvbuf, sizeof(recvbuf), 0);
recvbuf[rs] = '\0';
if (rs <= 0 || strncmp(recvbuf, "+OK", 3) != 0)
{
    return FALSE;
}
printf("%s", recvbuf);
return TRUE;
}

bool CPop3::List(int& sum)
{
/* 发送 LIST 命令 */
char sendbuf[128];
char recvbuf[256];
sprintf(sendbuf, "LIST\r\n");
send(m_sock, sendbuf, strlen(sendbuf), 0);
printf("%s", sendbuf);

int rs = Pop3Recv(recvbuf, sizeof(recvbuf), 0);
if (rs <= 0 || strncmp(recvbuf, "+OK", 3) != 0)
{
    return FALSE;
}
recvbuf[rs] = '\0';
printf("%s", recvbuf);
sum = GetMailSum(recvbuf);                          //得到邮件的数目
cout << sum << endl;
return TRUE;
}
bool CPop3::FetchEx(int num)
{
int rs;
FILE* fp;
int flag = 0;
unsigned int len;
char filename[256];
char sendbuf[128];
char recvbuf[20480];
/* 发送 RETR 命令 */
sprintf(sendbuf, "RETR %d\r\n", num);
send(m_sock, sendbuf, strlen(sendbuf), 0);
do
{
    rs = Pop3Recv(recvbuf, sizeof(recvbuf), 0);   //接收数据
    if (rs < 0)
```

```
    {
        return FALSE;
    }
    recvbuf[rs] = '\0';
    printf("Recv RETR Resp:\n %s", recvbuf);            //输出接收的数据
    if (flag == 0)
    {
        _itoa(num, filename, 10);                       //按照序号给文件排名
        strcat(filename, ".eml");

        flag = 1;
        fp = fopen(filename, "wb");                     //准备写文件
    }
    len = strlen(recvbuf);
    char * contentlen = new char[];                     //邮件内容信息的长度
    int contentlength;
    char md5val[33];                                    //取出的 MD5 的值
    md5val[32] = '\0';                                  //终结符
    memcpy(contentlen, recvbuf + len - 23 - 3, 3);      //取消息长度
    memcpy(md5val, recvbuf + len - 23 - 3 - 32, 32);    //取哈希值
    contentlength = atoi(contentlen);
    char * content = new char[];
    memcpy(content, recvbuf + len - 23 - 3 - 32 - contentlength, contentlength);    //取内容
    content[contentlength] = '\0';                      //把内容补充成为字符串

    unsigned char * final = NULL;
    char * iv = "wereachsuccessalready";
    unsigned char * i = (unsigned char * )iv;
    char * key = "wearethemoststronger";
    unsigned char * k = (unsigned char * )key;
    int ciphertext_len = 0;
    unsigned char ciphertext[MAX_CHAR_SIZE];
    unsigned char plaintext[MAX_CHAR_SIZE];
    final = decrypt_text(i, k, (unsigned char * )content, &contentlength, plaintext);
    content = (char * )final;
    content[strlen(content)] = '\0';
    if (strcmp(MD5(content), md5val) == 0)
        cout << "邮件内容完整性良好!" << endl;
    fwrite(recvbuf, 1, len - 23 - 3 - 32 - contentlength, fp);
    fwrite(content, 1, strlen(content), fp);
    fwrite(recvbuf + len - 23, 1, 24, fp);
    fflush(fp);                                         //刷新
} while (strstr(recvbuf, "\r\n.\r\n") == (char * )NULL);
fclose(fp);
return TRUE;
}
bool CPop3::Quit()
{
char sendbuf[128];
```

```
char recvbuf[128];

/* 发送 QUIT 命令 */
sprintf(sendbuf, "QUIT\r\n");
send(m_sock, sendbuf, strlen(sendbuf), 0);
int rs = recv(m_sock, recvbuf, sizeof(recvbuf), 0);
if (rs <= 0 || strncmp(recvbuf, "+OK", 3) != 0)
{
    return FALSE;
}
closesocket(m_sock);
return TRUE;
}

int CPop3::GetMailSum(char * buf)
{
int sum = 0;
char * p = strstr(buf, "\r\n");
if (p == NULL)
    return sum;
p = strstr(p + 2, "\r\n");
if (p == NULL)
    return sum;
while ((p = strstr(p + 2, "\r\n")) != NULL)
{
    sum++;
}

return sum;
}

/* main.cpp */
#include <stdio.h>
#include "pop3.h"
#include <iostream>
#include <string>
#include <string.h>
#include <conio.h>
#include <iostream>
#include <stdlib.h>
#include <openssl/md5.h>

using namespace std;
int main()
```

```
{
*/
cout << "--------------------登入邮箱-------------------------" << endl;
int i;
int sum;
string l1, l2, l3;
CPop3 pop3;
cout << "请输入服务器地址(pop):";
cin >> l3;
cout << "请输入用户名:";
cin >> l1;
cout << "请输入密码:";
char ch;
while ((ch = _getch()) != 13)
{
    l2 += ch;                        //string 对象重载了 +=
    cout << "*";
}
cout << endl;
char userName[256];
char password[256];
char srv[256];
for (i = 0; i < l3.length(); i++){
    srv[i] = l3[i];
}
srv[i] = '\0';
for (i = 0; i < l2.length(); i++){
    password[i] = l2[i];
}
password[i] = '\0';
for (i = 0; i < l1.length(); i++){
    userName[i] = l1[i];
}
userName[i] = '\0';
pop3.Create(userName, password, srv, 110);
pop3.Connect();                      //连接 pop3 服务器
pop3.Login();
pop3.List(sum);
if (sum < 0)
    printf("You have no letter in box !");
int n;
cout << "一共有" << sum << "封邮件,请输入你想查看第几封邮件: ";
cin >> n;
pop3.FetchEx(n);
pop3.Quit();

return 0;
}
```

运行结果如下。

1）发送邮件(图 8-6 与图 8-7)



图 8-6　邮件发送页面

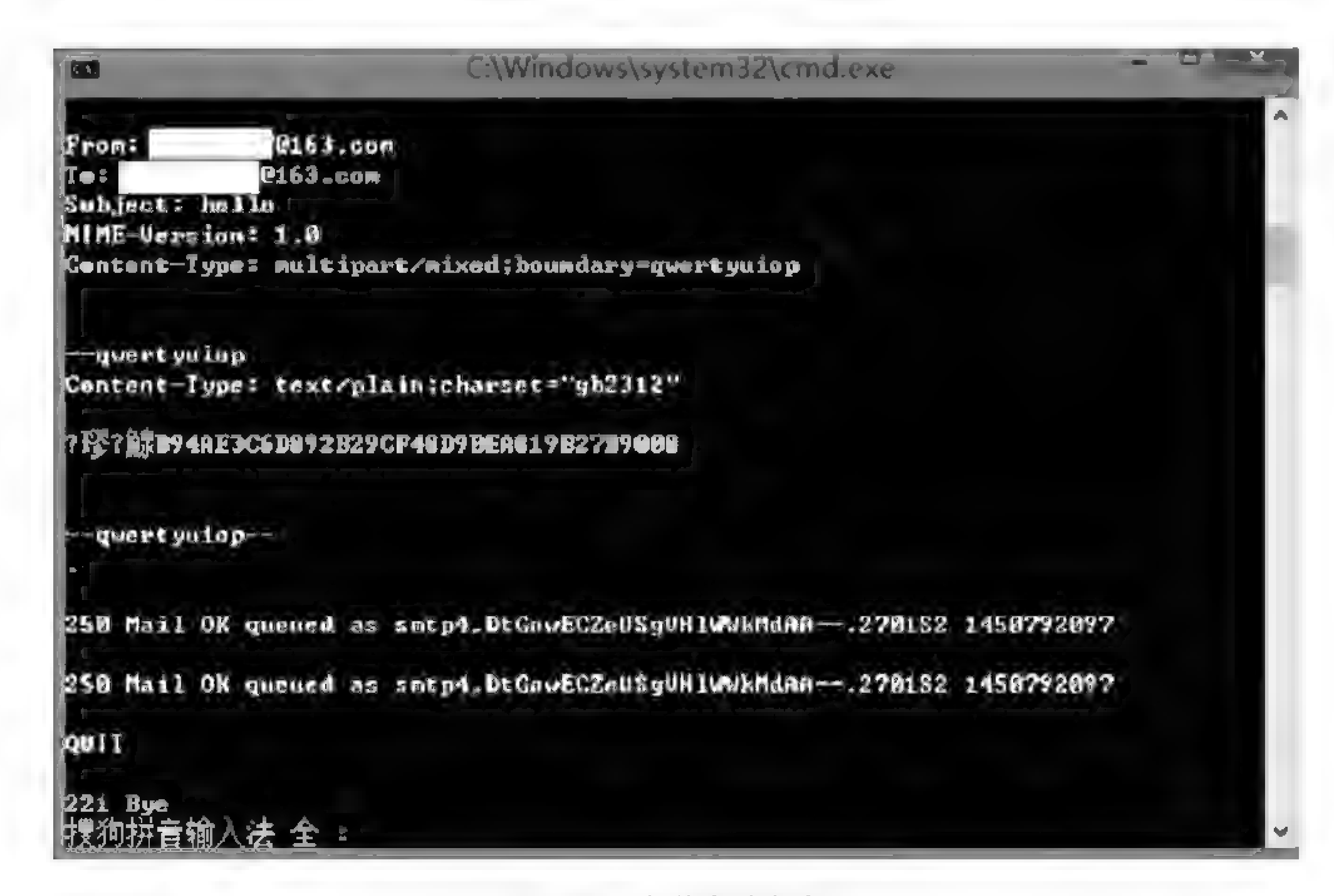


图 8-7　邮件发送成功

2）接收邮件（图 8-8～图 8-10）

```
C:\Windows\system32\cmd.exe
-------------------登入邮箱--------------------------
请输入服务器地址(pop):pop.163.com
请输入用户名:[redacted]@163.com
请输入密码:***********+OK Welcome to coremail Mail Pop3 Server (163coms[726cd87d72d896a1ac393507346040fas])
USER [redacted]@163.com
+OK core mail
PASS syeshuaiqi
+OK 5 message(s) [3993 byte(s)]
LIST
+OK 5 3993
1 805
2 799
3 798
4 793
5 798
.
5
一共有5封邮件，请输入你想查看第几封邮件：5
Recv RETR Resp:
 +OK 798 octets
Received: from [redacted]@163.com (unknown [223.166.63.99])
        by smtp4 (Coremail) with SMTP id DtGowECZeUSgVH1WWkMdAA  .278152;
        Tue, 22 Dec 2015 21:48:17 +0800 (CST)
搜狗拼音输入法 全 :com
```

图 8-8　邮件接收页面

```
C:\Windows\system32\cmd.exe
一共有5封邮件，请输入你想查看第几封邮件：5
Recv RETR Resp:
 +OK 798 octets
Received: from [redacted]@163.com (unknown [223.166.63.99])
        by smtp4 (Coremail) with SMTP id DtGowECZeUSgVH1WWkMdAA--.278152;
        Tue, 22 Dec 2015 21:48:17 +0800 (CST)
From: [redacted]@163.com
To: [redacted]@163.com
Subject: hello
MIME-Version: 1.0
Content-Type: multipart/mixed;boundary=qwertyuiop
X-CM-TRANSID:DtGowECZeUSgVH1WWkMdAA--.278152
Message-Id:<567954A1.023EE7.035480m50-134.163.com>
X-Coremail-Antispam: 1Uf129KBjDUn29KB7ZKAUJUUUUU529EdanIXcx71UUUUU7v73
        VFW2AGmfu7bjvjm3AaLaJ3UbIYCTnIWIevJa73UjIFyIuYvjxU2zBIUUUUU
X-Originating-IP: [223.166.63.99]
Date: Tue, 22 Dec 2015 21:48:17 +0800 (CST)
X-CM-SenderInfo: 520fgiqswxqiywtou0bp/xtbBzQHYw1O-xrrkkwAAsx

--qwertyuiop
Content-Type: text/plain;charset="gb2312"

?瑛?鯀B94AE3C6D892B29CF48D9BEA819B27B9008

搜狗拼音输入法 全 :
```

图 8-9　邮件接收过程

```
C:\Windows\system32\cmd.exe
Message-Id:<567954A1.023EE7.035480m50-134.163.com>
X-Coremail-Antispam: 1Uf129KBjDUn29KB7ZKAUJUUUUU529EdanIXcx71UUUUU7v73
        VFW2AGmfu7bjvjm3AaLaJ3UbIYCTnIWIevJa73UjIFyIuYvjxU2zBIUUUUU
X-Originating-IP: [223.166.63.99]
Date: Tue, 22 Dec 2015 21:48:17 +0800 (CST)
X-CM-SenderInfo: 520fgiqswxqiywtou0bp/xtbBzQHYw1O-xrrkkwAAsx

--qwertyuiop
Content-Type: text/plain;charset="gb2312"

?瑛?鯀B94AE3C6D892B29CF48D9BEA819B27B9008

--qwertyuiop--

邮件内容完整性良好!
请按任意键继续. . .
```

图 8-10　邮件接收成功

小　结

本章主要列举了安全编程的几个常见的应用程序，即基于OpenSSL的安全Web服务器和安全电子邮件传输。对各个应用程序的相关基础知识分别进行了简要介绍，并提供了相应的实现代码，为读者进一步加深理解安全编程提供良好的基础。

思　考　题

1. SSL协议是如何工作的?
2. HTTPS与HTTP的区别是什么?
3. 描述电子邮件的传输过程。
4. 思考Web服务器安全程序还有哪些实现方式，并尝试实现其中一种。
5. 安全电子邮件编程中如何实现对垃圾邮件的过滤? 请尝试实现。

参考文献

[1] 李晖,张宁,等.网络空间安全学科人才培养之思考[J].网络与信息安全学报,2015,(1).

[2] 李建华,邱卫东,孟魁,等.网络空间安全一级学科内涵建设和人才培养思考[J].信息安全研究,2015.

[3] 崔光耀,冯雪竹,等.强力推进网络空间安全一级学科建设——访沈昌祥院士[J].中国信息安全,2015.

[4] 李红娇,等.信息安全概论[M].2版.北京:中国电力出版社,2016.

[5] [美]William Stallings,[澳] Lawrie Brown.计算机安全——原理与实践[M].2版.王昭,等译.北京:电子工业出版社,2015.

[6] [美]William Stallings.密码编码学与网络安全——原理与实践[M].6版.唐明,等译.北京:电子工业出版社,2015.

[7] 陈卓,阮鸥,沈剑.网络安全编程与实践[M].北京:国防工业出版社,2008.

[8] 刘文涛.网络安全编程技术与实例[M].北京:机械工业出版社,2008.

[9] 吴功宜,张健忠,张健,等.网络安全高级软件编程技术[M].北京:清华大学出版社,2010.

[10] 李峰,陈向益,等.TCP/IP协议分析与应用编程[M].北京:人民邮电出版社,2008.

[11] 任泰明.TCP/IP网络编程[M].北京:人民邮电出版社,2009.

[12] Ofir Arkin. ICMP usage in Scanning[R/OL]. http://www.sys-security.com/architect/papers/icmp_scanning v3.0.pdf.

[13] Fyodor. Remote OS Detection via TCP/IP Fingerprinting (2nd Generation) [R/OL]. http://nmap.org/osdetect/.

[14] ICMP TYPE NUMBERS[R/OL]. http://www.iana.org/assignments/icmp-parameters.

[15] 连一峰,戴英侠,胡艳,等.分布式入侵检测模型研究[J].计算机研究与发展,2003,40(8).

[16] 刘春.基于组合算法选择特征的网络入侵检测模型[J].计算机与现代化,2014,(8).

[17] 李玉霞,刘丽,沈桂兰.基于AFSA-SVM的网络入侵检测模型[J].计算机工程与应用,2013,49(24).

[18] 魏群,刘玉芳,刘保相.电子商务中CA认证的OPENSSL实现方法[J].微计算机信息,2008,24(6).

[19] 刘维岗.基于IPv6的防火墙包过滤技术研究[J].计算机与现代化,2012,(3).

网络链接

[1] http://news.xinhuanet.com/politics/2015-06/04/c_127879542.htm,积极构建网络空间安全创新人才培养体系.

[2] http://www.myhack58.com/Article/html/3/68/2013/39849.html.

[3] http://xidong.net/File001/File_75831.html.

[4] http://blog.csdn.net/column/details/visualcppsafe.html.

[5] http://baike.baidu.com/link? url=Z_22_GylROZRUlPeSvPwik7nXPJIJOAF3mhjQsYt2UQ5KPRyAF-FrhWfpM4A1zl0yT7pcmc8PP9nZgjAqcw1Rq.

[6] http://blog.csdn.net/gdwzh/article/details/19230.

图书资源支持

感谢您一直以来对清华版图书的支持和爱护。为了配合本书的使用，本书提供配套的素材，有需求的用户请到清华大学出版社主页(http://www.tup.com.cn)上查询和下载，也可以拨打电话或发送电子邮件咨询。

如果您在使用本书的过程中遇到了什么问题，或者有相关图书出版计划，也请您发邮件告诉我们，以便我们更好地为您服务。

我们的联系方式：

地　　址：北京海淀区双清路学研大厦A座707

邮　　编：100084

电　　话：010－62770175－4604

资源下载：http://www.tup.com.cn

电子邮件：weijj@tup.tsinghua.edu.cn

QQ：883604(请写明您的单位和姓名)

扫一扫

资源下载、样书申请

新书推荐、技术交流

用微信扫一扫右边的二维码，即可关注清华大学出版社公众号“书圈”。